

Tensor Mathematics and Neural Networks Guide

Abdulla Alshamsi

November 2025

Preface / Overview

This book is created by Abdulla Alshamsi to document the journey of mastering tensor mathematics, linear algebra, statistics, probability, differential calculus, optimization, and neural network transformations using Python and PyTorch.

The main purpose of this work is to provide a structured, hands-on approach to learning advanced concepts in AI and deep learning, combining rigorous mathematical laws with practical coding exercises.

This project has taken several years of study and practice to compile all laws, examples, and code exercises systematically.

The Tracks (T00–T08) included in this book:

- T00 – Tensor Algebra Laws
- T01 – Vector Laws (1D)
- T02 – Matrix Laws (2D)
- T03 – Higher-Dimension Tensor Laws
- T04 – Tensor Shape Transformation Laws
- T05 – Differential Calculus Laws
- T06 – Optimization Laws
- T07 – Statistics Probability Laws
- T08 – Neural Network Transformation Laws

Each Track is designed to build progressively, starting from fundamental tensor operations (T00) up to neural network transformations (T08), providing both theoretical understanding and practical coding skills.

1 Tensor Algebra Laws

The purpose of **T00** is to introduce the fundamental laws of tensor algebra using Python, PyTorch, and CUDA. This track contains the most essential mathematical operations that form the foundation of all tensor work in deep learning, AI, and GPU computation.

In **T00**, each exercise focuses on a single mathematical law. The student reads the law, understands its meaning, and then writes the corresponding PyTorch–CUDA code.

This track includes the following laws:

- E00 – Tensor Addition Law
- E01 – Tensor Subtraction Law
- E02 – Tensor Hadamard Multiplication Law
- E03 – Tensor Division Law
- E04 – Tensor Power Law
- E05 – Tensor Square Root Law
- E06 – Tensor Absolute Value Law
- E07 – Tensor Negation Law
- E08 – Tensor Sign Law
- E09 – Tensor Clamp Law
- E10 – Tensor Round Law
- E11 – Tensor Floor Law
- E12 – Tensor Ceil Law
- E13 – Tensor Reciprocal Law
- E14 – Tensor Modulo Law
- E15 – Tensor Logical AND Law
- E16 – Tensor Logical OR Law
- E17 – Tensor Logical NOT Law
- E18 – Tensor Equality Law
- E19 – Tensor Comparison Law

By completing T00, the student will learn how to turn mathematical tensor laws into executable PyTorch–CUDA code, building the core skills needed for advanced tensor mathematics.

2 T00 – E00 : Tensor Addition Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad y = (y_1, y_2) \in \mathbb{R}^2$$

The tensor addition operator

$$+ : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined component-wise as:

$$x + y = (x_1 + y_1, x_2 + y_2)$$

This operation satisfies the vector space axioms:

$$\begin{aligned}
 x + y &= y + x && \text{(commutativity)} \\
 (x + y) + z &= x + (y + z) && \text{(associativity)} \\
 x + 0 &= x && \text{(identity)} \\
 x + (-x) &= 0 && \text{(inverse)}
 \end{aligned}$$

Implementation Principle

To compute the tensor addition numerically, each component of the input tensors is evaluated independently. Thus, for computational purposes:

$$z_i = x_i + y_i, \quad i = 1, 2$$

This rule generalizes naturally to tensors of arbitrary dimension because addition is performed element-wise over all corresponding positions in the tensor.

Programming Task

Implement the mathematical operator above by writing a Python function that receives two scalar values and returns their tensor addition result.

```
def E00(x1, y1):
    # implement the mathematical law
    return z
```

3 T00 – E01 : Tensor Subtraction Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad y = (y_1, y_2) \in \mathbb{R}^2$$

The tensor subtraction operator

$$- : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined component-wise as:

$$x - y = (x_1 - y_1, x_2 - y_2)$$

This operator follows directly from addition and additive inverses:

$$x - y = x + (-y)$$

The subtraction law satisfies:

$$\begin{aligned}
 x - 0 &= x && \text{(identity)} \\
 x - x &= 0 && \text{(self-cancellation)} \\
 (x - y) + y &= x && \text{(inverse consistency)}
 \end{aligned}$$

Implementation Principle

For numerical computation, each component of the tensors is processed independently. Thus, the subtraction rule becomes:

$$z_i = x_i - y_i, \quad i = 1, 2$$

This generalizes to any tensor shape because subtraction is performed element-wise on all corresponding positions in the input tensors.

Programming Task

Implement the mathematical subtraction operator by writing a Python function that takes two scalar inputs and returns their tensor subtraction result.

```
def E01(x1, y1):
    # implement the mathematical law
    return z
```

4 T00 – E02 : Tensor Hadamard Multiplication Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad y = (y_1, y_2) \in \mathbb{R}^2$$

The Hadamard (element-wise) multiplication operator is the function:

$$\odot : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

It is defined component-wise as:

$$x \odot y = (x_1 y_1, x_2 y_2)$$

This operation satisfies:

$x \odot y = y \odot x$	(commutativity)
$(x \odot y) \odot z = x \odot (y \odot z)$	(associativity)
$x \odot (1, 1) = x$	(identity)
$x \odot (0, 0) = (0, 0)$	(annihilation)

Implementation Principle

For numerical computation, each component of the result is obtained by multiplying the corresponding elements of the input tensors:

$$z_i = x_i \cdot y_i, \quad i = 1, 2$$

This rule extends naturally to tensors of arbitrary dimension, since Hadamard multiplication always applies element-wise across matching positions in the tensors.

Implementation Principle

For numerical computation, each component of the result is obtained by multiplying the corresponding elements of the input tensors:

$$z_i = x_i \cdot y_i, \quad i = 1, 2$$

This rule extends naturally to tensors of arbitrary dimension, since Hadamard multiplication always applies element-wise across matching positions in the tensors.

Programming Task

Write a Python function that takes two scalar inputs and returns their Hadamard multiplication result.

```
def E02(x1, y1):
    # implement the mathematical law
    return z
```

5 T00 – E03 : Tensor Division Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad y = (y_1, y_2) \in \mathbb{R}^2 \setminus \{(0, 0)\}$$

The tensor division operator

$$\oslash : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined component-wise by:

$$x \oslash y = \left(\frac{x_1}{y_1}, \frac{x_2}{y_2} \right)$$

Division is only valid when every component of y is non-zero:

$$y_i \neq 0 \quad \text{for all } i$$

This operator satisfies:

$$\begin{aligned} x \oslash (1, 1) &= x && \text{(identity)} \\ x \oslash x &= (1, 1) && \text{(self-division)} \\ (x \oslash y) \odot y &= x && \text{(inverse consistency)} \end{aligned}$$

Implementation Principle

The computation is performed element-wise:

$$z_i = \frac{x_i}{y_i}, \quad i = 1, 2$$

For numerical stability, tensor division must avoid division by zero. In PyTorch, this can be handled by checking:

$$y_i \neq 0$$

This rule extends directly to tensors of any dimension.

Programming Task

Write a Python function that takes two scalar inputs and returns their tensor division result.

```
def E03(x1, y1):
    # implement the mathematical law
    return z
```

6 T00 – E04 : Tensor Power Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad y = (y_1, y_2) \in \mathbb{R}^2$$

The tensor power operator

$$\circledast : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined element-wise as:

$$x \circledast y = (x_1^{y_1}, x_2^{y_2})$$

This operation requires:

$$x_i > 0 \quad \text{whenever} \quad y_i \notin \mathbb{Z}$$

The power law satisfies:

$$\begin{aligned} x^{(1,1)} &= x && \text{(identity)} \\ x^{(0,0)} &= (1, 1) && \text{(zero exponent)} \\ (x \circledast y) \circledast z &= x \circledast (y \odot z) && \text{(associativity of exponents)} \\ x \circledast (y + z) &= (x \circledast y) \odot (x \circledast z) && \text{(exponent distribution)} \end{aligned}$$

Implementation Principle

Tensor power is computed element-wise:

$$z_i = x_i^{y_i}, \quad i = 1, 2$$

This extends directly to tensors of any dimension. Care must be taken when:

$$x_i = 0 \quad \text{or} \quad x_i < 0$$

because certain exponent values may be undefined or lead to NaN.

Programming Task

Write a Python function that takes two scalar inputs and returns their tensor power result.

```
def E04(x, y):
    # apply the power law element-wise
    # for example, compute x ** y
    z = x ** y
    return z
```

7 T00 – E05 : Tensor Square Root Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}_{\geq 0}^2$$

The tensor square root operator

$$\sqrt{} : \mathbb{R}_{\geq 0}^2 \rightarrow \mathbb{R}_{\geq 0}^2$$

is defined component-wise as:

$$\sqrt{x} = (\sqrt{x_1}, \sqrt{x_2})$$

The square root satisfies the following identities:

$$\begin{aligned} \sqrt{x \odot x} &= x && \text{(inverse of squaring)} \\ \sqrt{x \odot y} &= \sqrt{x} \odot \sqrt{y} && \text{(multiplicative property)} \\ \sqrt{1} &= 1, \quad \sqrt{0} = 0 && \\ \sqrt{x} &\geq 0 && \text{(non-negativity)} \end{aligned}$$

Implementation Principle

The square root operation is applied independently to each component:

$$z_i = \sqrt{x_i}, \quad i = 1, 2$$

The input tensor must satisfy:

$$x_i \geq 0$$

because the real square root function is not defined for negative values.

This element-wise rule generalizes to any tensor dimension.

Programming Task

Write a Python function that computes the element-wise square root of a tensor input.

```
def E05(x):
    # compute the element-wise square root
    z = x ** 0.5
    return z
```

8 T00 – E06 : Tensor Absolute Value Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2$$

The tensor absolute value operator

$$|\cdot| : \mathbb{R}^2 \rightarrow \mathbb{R}_{\geq 0}^2$$

is defined element-wise as:

$$|x| = (|x_1|, |x_2|)$$

The absolute value satisfies the following classical identities:

$ x_i = \sqrt{x_i^2}$	(definition)
$ x_i \geq 0$	(non-negativity)
$ x_i = 0 \iff x_i = 0$	(zero property)
$ x_i y_i = x_i y_i $	(multiplicativity)
$ x_i + y_i \leq x_i + y_i $	(triangle inequality)

Thus, the tensor absolute value is computed by applying the scalar absolute value to each component.

Implementation Principle

The computation rule is:

$$z_i = |x_i|, \quad i = 1, 2$$

This extends directly to any tensor dimension since the absolute value acts independently on each element.

Programming Task

Write a Python function that computes the element-wise absolute value of a tensor input.

```
def E06(x):
    # apply the absolute value element-wise
    z = abs(x)
    return z
```

9 T00 – E07 : Tensor Negation Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2$$

The tensor negation operator

$$- : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined element-wise by:

$$-x = (-x_1, -x_2)$$

Negation satisfies:

$-(-x) = x$	(involution)
$x + (-x) = (0, 0)$	(additive inverse)
$-(x + y) = (-x) + (-y)$	(distribution)

Implementation Principle

Element-wise negation:

$$z_i = -x_i, \quad i = 1, 2$$

Programming Task

```
def E07(x):
    z = -x
    return z
```

10 T00 – E08 : Tensor Sign Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2$$

The sign operator

$$\text{sign} : \mathbb{R}^2 \rightarrow \{-1, 0, 1\}^2$$

is defined by:

$$\text{sign}(x_i) = \begin{cases} -1 & x_i < 0, \\ 0 & x_i = 0, \\ 1 & x_i > 0 \end{cases}$$

Thus:

$$\text{sign}(x) = (\text{sign}(x_1), \text{sign}(x_2))$$

Implementation Principle

Element-wise rule:

$$z_i = \text{sign}(x_i)$$

Programming Task

```
def E08(x):
    # sign function (manual or torch.sign)
    return (1 if x > 0 else -1 if x < 0 else 0)
```

11 T00 – E09 : Tensor Clamp Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad a, b \in \mathbb{R}, \quad a < b$$

The clamp operator

$$\text{clamp}_{[a,b]} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

is defined element-wise as:

$$\text{clamp}_{[a,b]}(x_i) = \begin{cases} a & x_i < a, \\ x_i & a \leq x_i \leq b, \\ b & x_i > b \end{cases}$$

$$\text{clamp}(x) = (\text{clamp}(x_1), \text{clamp}(x_2))$$

Implementation Principle

Element-wise:

$$z_i = \min(\max(x_i, a), b)$$

Programming Task

```
def E09(x, a, b):
    # clamp value between a and b
    return min(max(x, a), b)
```

12 T00 – E10 : Tensor Round Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2$$

The rounding operator

$$\text{round} : \mathbb{R}^2 \rightarrow \mathbb{Z}^2$$

is defined as:

$$\text{round}(x) = (\text{round}(x_1), \text{round}(x_2))$$

Where for each component:

$$\text{round}(x_i) = \arg \min_{n \in \mathbb{Z}} |x_i - n|$$

Rounding satisfies:

$$x_i - \frac{1}{2} < \text{round}(x_i) \leq x_i + \frac{1}{2}$$

Implementation Principle

Element-wise rule:

$$z_i = \text{nearest integer to } x_i$$

Programming Task

```
def E10(x):
    # nearest integer rounding
    return round(x)
```

13 T00 – E11 : Tensor Floor Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2.$$

The floor operator

$$\lfloor \cdot \rfloor : \mathbb{R}^2 \rightarrow \mathbb{Z}^2$$

is defined element-wise as:

$$\lfloor x \rfloor = (\lfloor x_1 \rfloor, \lfloor x_2 \rfloor)$$

where each component satisfies:

$$\lfloor x_i \rfloor \leq x_i < \lfloor x_i \rfloor + 1.$$

Properties of the floor function:

$$\begin{aligned} \lfloor x_i + n \rfloor &= \lfloor x_i \rfloor + n && \text{for } n \in \mathbb{Z} \\ \lfloor -x_i \rfloor &= -\lceil x_i \rceil \\ x_i - 1 < \lfloor x_i \rfloor &\leq x_i \end{aligned}$$

Implementation Principle

The floor operation acts independently on each element:

$$z_i = \lfloor x_i \rfloor, \quad i = 1, 2$$

This generalizes directly to tensors of any dimension.

Programming Task

Write a Python function that returns the floor value of a scalar input.

```
def E11(x):
    # floor: greatest integer <= x
    z = int(x // 1)
    return z
```

14 T00 – E12 : Tensor Ceil Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2.$$

The ceiling operator

$$\lceil \cdot \rceil : \mathbb{R}^2 \rightarrow \mathbb{Z}^2$$

is defined element-wise by:

$$\lceil x \rceil = (\lceil x_1 \rceil, \lceil x_2 \rceil)$$

where each component satisfies:

$$\lceil x_i \rceil - 1 < x_i \leq \lceil x_i \rceil.$$

The ceiling function has the following properties:

$$\begin{aligned} \lceil x_i + n \rceil &= \lceil x_i \rceil + n && \text{for } n \in \mathbb{Z} \\ \lceil -x_i \rceil &= -\lfloor x_i \rfloor \\ x_i &\leq \lceil x_i \rceil < x_i + 1 \end{aligned}$$

Implementation Principle

The computation is element-wise:

$$z_i = \lceil x_i \rceil, \quad i = 1, 2$$

Programming Task

Write a Python function that computes the ceiling (smallest integer x).

```
def E12(x):
    # ceil: smallest integer >= x
    z = int(-(-x // 1))  # mathematical ceiling
    return z
```

15 T00 – E13 : Tensor Reciprocal Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad x_i \neq 0.$$

The reciprocal operator

$$(\cdot)^{-1} : \mathbb{R}^2 \setminus \{0\}^2 \rightarrow \mathbb{R}^2$$

is defined element-wise as:

$$x^{-1} = \left(\frac{1}{x_1}, \frac{1}{x_2} \right)$$

It satisfies the identities:

$x \odot x^{-1} = (1, 1)$	(inverse property)
$(x \odot y)^{-1} = x^{-1} \odot y^{-1}$	(multiplicativity)
$(x^{-1})^{-1} = x$	(involution)

Implementation Principle

Reciprocal is applied independently to each component:

$$z_i = \frac{1}{x_i}, \quad i = 1, 2$$

The operator is undefined when:

$$x_i = 0.$$

Programming Task

Write a Python function that computes the reciprocal of a scalar input.

```
def E13(x):
    # reciprocal: 1 / x
    z = 1 / x
    return z
```

16 T00 – E14 : Tensor Modulo Law

Mathematical Definition

Let

$$x = (x_1, x_2) \in \mathbb{R}^2, \quad m \in \mathbb{Z}, \quad m > 0.$$

The modulo operator

$$\text{mod} : \mathbb{R}^2 \times \mathbb{Z}_{>0} \rightarrow \mathbb{R}^2$$

is defined element-wise as:

$$x \text{ mod } m = (x_1 \text{ mod } m, x_2 \text{ mod } m)$$

For each component x_i , the modulo satisfies the division identity:

$$x_i = qm + r, \quad 0 \leq r < m,$$

where

$$r = x_i \text{ mod } m.$$

The modulo operation satisfies:

$(x_i + km) \text{ mod } m = x_i \text{ mod } m$	(periodicity)
$(x_i \text{ mod } m) < m$	(range constraint)

Implementation Principle

The modulo computation is element-wise:

$$z_i = x_i \bmod m, \quad i = 1, 2.$$

The result always lies in the interval:

$$0 \leq z_i < m.$$

Programming Task

Write a Python function that computes the modulo of a scalar with respect to m .

```
def E14(x, m):
    # modulo: remainder after division
    z = x % m
    return z
```

17 T00 – E15 : Tensor Logical AND Law

Mathematical Definition

Let

$$x = (x_1, x_2), \quad y = (y_1, y_2)$$

where

$$x_i, y_i \in \{0, 1\}.$$

The logical AND operator

$$\wedge : \{0, 1\}^2 \times \{0, 1\}^2 \rightarrow \{0, 1\}^2$$

is defined element-wise as:

$$x \wedge y = (x_1 \wedge y_1, x_2 \wedge y_2)$$

The scalar AND truth rules are:

$$a \wedge b = \begin{cases} 1 & \text{if } a = 1 \text{ and } b = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Thus the tensor satisfies:

$$(x \wedge y)_i = \begin{cases} 1 & x_i = 1 \text{ and } y_i = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Properties:

$$\begin{aligned} x \wedge y &= y \wedge x && \text{(commutativity)} \\ x \wedge (y \wedge z) &= (x \wedge y) \wedge z && \text{(associativity)} \\ x \wedge (1, 1) &= x && \text{(identity)} \\ x \wedge (0, 0) &= (0, 0) && \text{(annihilation)} \end{aligned}$$

Implementation Principle

Element-wise evaluation:

$$z_i = \begin{cases} 1 & x_i = 1 \wedge y_i = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Programming Task

Write a Python function that computes the logical AND of two scalar inputs.

```
def E15(a, b):
    # logical AND
    z = int(bool(a and b))
    return z
```

18 T00 – E16 : Tensor Logical OR Law

Mathematical Definition

Let

$$x = (x_1, x_2), \quad y = (y_1, y_2)$$

where

$$x_i, y_i \in \{0, 1\}.$$

The logical OR operator

$$\vee : \{0, 1\}^2 \times \{0, 1\}^2 \rightarrow \{0, 1\}^2$$

is defined element-wise as:

$$x \vee y = (x_1 \vee y_1, \quad x_2 \vee y_2)$$

The scalar OR truth rule is:

$$a \vee b = \begin{cases} 1 & \text{if } a = 1 \text{ or } b = 1, \\ 0 & \text{if } a = 0 \text{ and } b = 0. \end{cases}$$

Thus the tensor satisfies:

$$(x \vee y)_i = \begin{cases} 1 & x_i = 1 \text{ or } y_i = 1, \\ 0 & x_i = 0 \text{ and } y_i = 0. \end{cases}$$

Properties:

$x \vee y = y \vee x$	(commutativity)
$x \vee (y \vee z) = (x \vee y) \vee z$	(associativity)
$x \vee (0, 0) = x$	(identity)
$x \vee (1, 1) = (1, 1)$	(absorption)

Implementation Principle

Element-wise rule:

$$z_i = \begin{cases} 1 & x_i = 1 \text{ or } y_i = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Programming Task

Write a Python function that computes the logical OR of two scalar inputs.

```
def E16(a, b):
    # logical OR
    z = int(bool(a or b))
    return z
```

19 T00 – E17 : Tensor Logical NOT Law

Mathematical Definition

Let

$$x = (x_1, x_2), \quad x_i \in \{0, 1\}.$$

The logical negation operator

$$\neg : \{0, 1\}^2 \rightarrow \{0, 1\}^2$$

is defined element-wise as:

$$\neg x = (\neg x_1, \quad \neg x_2)$$

The scalar negation rule is:

$$\neg a = \begin{cases} 1 & \text{if } a = 0, \\ 0 & \text{if } a = 1. \end{cases}$$

Thus the tensor satisfies:

$$(\neg x)_i = \begin{cases} 1 & x_i = 0, \\ 0 & x_i = 1. \end{cases}$$

Properties:

$\neg(\neg x) = x$	(involution)
$x \vee \neg x = (1, 1)$	(excluded middle)
$x \wedge \neg x = (0, 0)$	(non-contradiction)

Implementation Principle

Apply logical negation element-wise:

$$z_i = \begin{cases} 1 & x_i = 0, \\ 0 & x_i = 1. \end{cases}$$

Programming Task

Write a Python function that computes the logical NOT of a scalar input.

```
def E17(a):
    # logical NOT
    z = int(not a)
    return z
```

20 T00 – E18 : Tensor Equality Law

Mathematical Definition

Let

$$x = (x_1, x_2), \quad y = (y_1, y_2).$$

The equality operator

$$\text{eq} : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \{0, 1\}^2$$

is defined element-wise as:

$$\text{eq}(x, y) = ([x_1 = y_1], [x_2 = y_2])$$

where the indicator notation $[P]$ returns:

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true,} \\ 0 & \text{if } P \text{ is false.} \end{cases}$$

Thus each component satisfies:

$$\text{eq}(x, y)_i = \begin{cases} 1 & x_i = y_i, \\ 0 & x_i \neq y_i. \end{cases}$$

Equality properties:

$x = x$	(reflexivity)
$x = y \Rightarrow y = x$	(symmetry)
$x = y, y = z \Rightarrow x = z$	(transitivity)

Implementation Principle

Element-wise comparison:

$$z_i = [x_i = y_i], \quad i = 1, 2.$$

Programming Task

Write a Python function that checks equality of two scalar inputs.

```
def E18(a, b):
    # equality test
    z = int(a == b)
    return z
```

21 T00 – E19 : Tensor Comparison Law

Mathematical Definition

Let

$$x = (x_1, x_2), \quad y = (y_1, y_2).$$

Tensor comparison defines three element-wise relational operators:

$$>, <, ==$$

Each operator returns a Boolean tensor in $\{0, 1\}^2$.

$$x > y = ([x_1 > y_1], [x_2 > y_2])$$

$$x < y = ([x_1 < y_1], [x_2 < y_2])$$

$$x == y = ([x_1 == y_1], [x_2 == y_2])$$

Using indicator notation:

$$[P] = \begin{cases} 1 & \text{if predicate } P \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Properties:

$$\begin{aligned} x_i > y_i &\Rightarrow y_i < x_i \\ x_i = y_i &\Rightarrow (x_i \not> y_i) \wedge (x_i \not< y_i) \\ x_i > y_i \vee x_i = y_i \vee x_i < y_i &= 1 \quad (\text{trichotomy}) \end{aligned}$$

Implementation Principle

Element-wise rule:

$$z_i = [x_i > y_i], \quad w_i = [x_i < y_i], \quad u_i = [x_i == y_i].$$

Programming Task

Write a Python function that returns 1 if $a > b$, and 0 otherwise.

```
def E19(a, b):
    # comparison: a > b
    z = int(a > b)
    return z
```

22 Vector Laws (1D)

The purpose of **T01** is to introduce the fundamental laws of **1-dimensional vector mathematics** using Python, PyTorch, and CUDA. This track focuses on the essential operations that define vectors in \mathbb{R}^n and builds the foundation for understanding higher-dimensional tensor operations used in deep learning, AI models, and GPU computation.

In **T01**, each exercise presents one mathematical law. The student reads the vector formula, understands the mathematical meaning, and then implements the law directly using PyTorch–CUDA.

This track includes the following laws:

- E00 – 1D Dot Product Law
- E01 – 1D Cross Product Law
- E02 – 1D Norm Law
- E03 – 1D Normalize Law
- E04 – 1D Projection Law
- E05 – 1D Angle Law
- E06 – 1D Distance Law
- E07 – 1D Sum Reduction Law
- E08 – 1D Max Reduction Law
- E09 – 1D Min Reduction Law
- E10 – 1D Mean Reduction Law
- E11 – 1D Variance Law
- E12 – 1D Standard Deviation Law
- E13 – 1D Cumulative Sum Law
- E14 – 1D Cumulative Product Law

By completing **T01**, the student will gain a strong understanding of vector algebra and how to convert vector mathematical laws into efficient PyTorch–CUDA code. This knowledge is critical for progressing into matrix operations (T02), higher-dimensional tensors (T03), and neural network computation.

23 T01 – E00 : 1D Dot Product Law

Mathematical Definition

Let two vectors

$$x = (x_1, x_2, \dots, x_n), \quad y = (y_1, y_2, \dots, y_n)$$

be elements of \mathbb{R}^n .

The **dot product** is a mapping

$$\cdot : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$$

defined by:

$$x \cdot y = \sum_{i=1}^n x_i y_i$$

The dot product satisfies the following properties:

$x \cdot y = y \cdot x$	(commutativity)
$x \cdot (y + z) = x \cdot y + x \cdot z$	(distributivity)
$(\alpha x) \cdot y = \alpha(x \cdot y)$	(scalar associativity)
$x \cdot x = \ x\ ^2$	(relation to norm)

The dot product measures how aligned two vectors are.

Implementation Principle

Element-wise multiplication followed by summation:

$$x \cdot y = (x_1 y_1) + (x_2 y_2) + \cdots + (x_n y_n)$$

Programming Task

Write a Python function that computes the dot product of two vectors.

```
def E00(x, y):
    # 1D dot product
    z = 0
    for i in range(len(x)):
        z += x[i] * y[i]
    return z
```

24 T01 – E01 : 1D Cross Product Law

Mathematical Definition

The cross product is defined only for 3-dimensional vectors.

Let

$$x = (x_1, x_2, x_3), \quad y = (y_1, y_2, y_3) \in \mathbb{R}^3.$$

The **cross product** is a mapping

$$\times : \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$$

defined by:

$$x \times y = \begin{pmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{pmatrix}$$

Geometric interpretation:

$$\|x \times y\| = \|x\| \|y\| \sin(\theta)$$

where θ is the angle between x and y .

Properties:

$x \times y = -(y \times x)$	(anti-commutative)
$x \times (y + z) = (x \times y) + (x \times z)$	(distributive)
$x \times x = (0, 0, 0)$	(self-cross is zero)
$x \cdot (x \times y) = 0$	(orthogonality)

Implementation Principle

Apply the determinant pattern:

$$\begin{aligned} z_1 &= x_2 y_3 - x_3 y_2 \\ z_2 &= x_3 y_1 - x_1 y_3 \\ z_3 &= x_1 y_2 - x_2 y_1 \end{aligned}$$

Programming Task

Write a Python function that computes the cross product of two 3D vectors.

```
def E01(x, y):
    # 1D cross product for 3-element vectors
    z1 = x[1] * y[2] - x[2] * y[1]
    z2 = x[2] * y[0] - x[0] * y[2]
    z3 = x[0] * y[1] - x[1] * y[0]
    return (z1, z2, z3)
```

25 T01 – E02 : 1D Norm Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **norm** (also called magnitude or length) is the mapping

$$\|\cdot\| : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$$

defined by:

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

This is the standard Euclidean (L2) norm.

Properties:

$\ x\ \geq 0$	(non-negativity)
$\ x\ = 0 \iff x = 0$	(definiteness)
$\ \alpha x\ = \alpha \ x\ $	(homogeneity)
$\ x + y\ \leq \ x\ + \ y\ $	(triangle inequality)

Relationship to dot product:

$$\|x\| = \sqrt{x \cdot x}$$

Implementation Principle

Compute the sum of squares and apply the square root:

$$\|x\| = \sqrt{\sum x_i^2}$$

Programming Task

Write a Python function that computes the L2 norm of a vector.

```
def E02(x):
    # L2 norm
    s = 0
    for i in range(len(x)):
        s += x[i] * x[i]
    return s ** 0.5
```

26 T01 – E03 : 1D Normalize Law

Mathematical Definition

Let

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n, \quad x \neq 0.$$

The **normalization** of a vector is the mapping

$$\text{normalize} : \mathbb{R}^n \setminus \{0\} \rightarrow \mathbb{R}^n$$

defined by:

$$\hat{x} = \frac{x}{\|x\|}$$

where

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

The normalized vector \hat{x} satisfies:

$$\|\hat{x}\| = 1$$

Properties:

$$\begin{aligned} \frac{x}{\|x\|} \cdot \frac{x}{\|x\|} &= 1 && \text{(unit vector)} \\ \text{normalize}(\alpha x) &= \text{normalize}(x) && \text{(scale-invariance)} \\ x &= \|x\| \hat{x} && \text{(polar decomposition)} \end{aligned}$$

Implementation Principle

Apply:

$$\hat{x}_i = \frac{x_i}{\|x\|}$$

Programming Task

Write a Python function that normalizes a vector.

```
def E03(x):
    # normalization
    n = 0
    for i in range(len(x)):
        n += x[i] * x[i]
    n = n ** 0.5      # compute norm

    # divide each component by the norm
    out = []
    for i in range(len(x)):
        out.append(x[i] / n)
    return out
```

27 T01 – E04 : 1D Projection Law

Mathematical Definition

Let two non-zero vectors

$$x = (x_1, x_2, \dots, x_n), \quad y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n.$$

The **projection of x onto y** is the mapping

$$\text{proj}_y(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

defined by:

$$\text{proj}_y(x) = \left(\frac{x \cdot y}{\|y\|^2} \right) y$$

Where:

$$\begin{aligned} x \cdot y &= \sum_{i=1}^n x_i y_i \\ \|y\|^2 &= y_1^2 + y_2^2 + \dots + y_n^2 \end{aligned}$$

Geometric interpretation:

$$\text{proj}_y(x) = (\text{length of } x \text{ along } y) \cdot \text{unit direction of } y$$

Properties:

$$\begin{aligned} \text{proj}_y(x) &= x && \text{if } x \parallel y \\ \text{proj}_y(x) &= 0 && \text{if } x \perp y \\ \text{proj}_y(\alpha x) &= \alpha \text{proj}_y(x) \end{aligned}$$

Implementation Principle

Compute:

$$\text{proj}_y(x)_i = \left(\frac{x \cdot y}{\|y\|^2} \right) y_i$$

Programming Task

Write a Python function that computes the projection of vector x onto vector y .

```
def E04(x, y):
    # compute dot product x·y
    dot = 0
    for i in range(len(x)):
        dot += x[i] * y[i]

    # compute ||y||^2
    norm_y_sq = 0
    for i in range(len(y)):
        norm_y_sq += y[i] * y[i]

    # scaling factor
    scale = dot / norm_y_sq

    # projection result
    out = []
    for i in range(len(y)):
        out.append(scale * y[i])

    return out
```

28 T01 – E05 : 1D Angle Law

Mathematical Definition

Let two non-zero vectors

$$x = (x_1, x_2, \dots, x_n), \quad y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n.$$

The **angle** θ between x and y is defined by:

$$\cos(\theta) = \frac{x \cdot y}{\|x\| \|y\|}$$

Thus,

$$\theta = \arccos \left(\frac{x \cdot y}{\|x\| \|y\|} \right)$$

Where:

$$\begin{aligned} x \cdot y &= \sum_{i=1}^n x_i y_i \\ \|x\| &= \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \\ \|y\| &= \sqrt{y_1^2 + y_2^2 + \dots + y_n^2} \end{aligned}$$

Geometric interpretation:

- $\theta = 0^\circ$ if x and y point in the same direction.
- $\theta = 90^\circ$ if x and y are orthogonal.

- $\theta = 180^\circ$ if they point in opposite directions.

Properties:

$$x \cdot y = \|x\| \|y\| \cos(\theta)$$

$$-1 \leq \cos(\theta) \leq 1$$

Implementation Principle

Compute:

$$\theta = \arccos \left(\frac{\sum x_i y_i}{\sqrt{\sum x_i^2} \sqrt{\sum y_i^2}} \right)$$

Programming Task

Write a Python function that computes the angle between two vectors.

```
import math

def E05(x, y):
    # dot product
    dot = 0
    for i in range(len(x)):
        dot += x[i] * y[i]

    # norms
    norm_x = 0
    norm_y = 0
    for i in range(len(x)):
        norm_x += x[i] * x[i]
        norm_y += y[i] * y[i]

    norm_x = math.sqrt(norm_x)
    norm_y = math.sqrt(norm_y)

    # cosine(theta)
    cos_theta = dot / (norm_x * norm_y)

    # angle
    theta = math.acos(cos_theta)
    return theta
```

29 T01 – E06 : 1D Distance Law

Mathematical Definition

Let two vectors

$$x = (x_1, x_2, \dots, x_n), \quad y = (y_1, y_2, \dots, y_n) \in \mathbb{R}^n.$$

The **distance** between x and y is defined by the Euclidean formula:

$$d(x, y) = \|x - y\|$$

Where the vector difference is:

$$x - y = (x_1 - y_1, x_2 - y_2, \dots, x_n - y_n)$$

Thus the distance becomes:

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

Equivalent dot-product form:

$$d(x, y) = \sqrt{(x - y) \cdot (x - y)}$$

Properties:

$d(x, y) \geq 0$	(non-negativity)
$d(x, y) = 0 \iff x = y$	(identity)
$d(x, y) = d(y, x)$	(symmetry)
$d(x, z) \leq d(x, y) + d(y, z)$	(triangle inequality)

Implementation Principle

Compute:

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Programming Task

Write a Python function that computes the Euclidean distance.

```
def E06(x, y):
    # Euclidean distance
    s = 0
    for i in range(len(x)):
        diff = x[i] - y[i]
        s += diff * diff
    return s ** 0.5
```

30 T01 – E07 : 1D Sum Reduction Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **sum reduction** is the mapping

$$\text{sum} : \mathbb{R}^n \rightarrow \mathbb{R}$$

defined by:

$$\text{sum}(x) = \sum_{i=1}^n x_i$$

The sum reduction satisfies:

$\text{sum}(x + y) = \text{sum}(x) + \text{sum}(y)$	(linearity)
$\text{sum}(\alpha x) = \alpha \text{sum}(x)$	(homogeneity)
$\text{sum}(x) = x_1 + x_2 + \dots + x_n$	(definition)

The sum reduction collapses a vector into a single scalar value.

Implementation Principle

Compute:

$$s = x_1 + x_2 + \dots + x_n$$

Programming Task

Write a Python function that computes the sum of a vector.

```
def E07(x):
    s = 0
    for i in range(len(x)):
        s += x[i]
    return s
```

31 T01 – E08 : 1D Max Reduction Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **maximum reduction** is the mapping

$$\max : \mathbb{R}^n \rightarrow \mathbb{R}$$

defined by:

$$\max(x) = \max\{x_1, x_2, \dots, x_n\}$$

This returns the largest component of the vector.

The max reduction satisfies:

$$\begin{aligned}\max(x) &\geq x_i, \quad \forall i \\ \max(x) &= x_k \quad \text{for some index } k \\ \max(x + c) &= \max(x) + c \quad \text{for scalar } c\end{aligned}$$

Max reduction is frequently used in neural networks, especially in:

- softmax computations,
- normalization,
- numerical stability operations.

Implementation Principle

Compute:

$$\max(x) = \text{largest value among } x_1, x_2, \dots, x_n$$

Programming Task

Write a Python function that returns the maximum value in a vector.

```
def E08(x):
    m = x[0]
    for i in range(1, len(x)):
        if x[i] > m:
            m = x[i]
    return m
```

32 T01 – E09 : 1D Min Reduction Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **minimum reduction** is the mapping

$$\min : \mathbb{R}^n \rightarrow \mathbb{R}$$

defined by:

$$\min(x) = \min\{x_1, x_2, \dots, x_n\}$$

This returns the smallest element of the vector.

The min reduction satisfies:

$$\begin{aligned}\min(x) &\leq x_i & \forall i \\ \min(x) &= x_k & \text{for some index } k \\ \min(x + c) &= \min(x) + c & \text{for scalar } c\end{aligned}$$

The minimum operator is fundamental in:

- optimization,
- thresholding,
- activation functions,
- bounding and constraint operations.

Implementation Principle

Compute:

$$\min(x) = \text{smallest value among } x_1, x_2, \dots, x_n$$

Programming Task

Write a Python function that returns the minimum value in a vector.

```
def E09(x):
    m = x[0]
    for i in range(1, len(x)):
        if x[i] < m:
            m = x[i]
    return m
```

33 T01 – E10 : 1D Mean Reduction Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **mean** (average) of the vector is the mapping

$$\text{mean} : \mathbb{R}^n \rightarrow \mathbb{R}$$

defined by:

$$\text{mean}(x) = \frac{1}{n} \sum_{i=1}^n x_i$$

This is also written as:

$$\mu = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Properties:

$\min(x) \leq \text{mean}(x) \leq \max(x)$	(boundedness)
$\text{mean}(x + c) = \text{mean}(x) + c$	(translation)
$\text{mean}(\alpha x) = \alpha \text{mean}(x)$	(scaling)
$\text{sum}(x) = n \cdot \text{mean}(x)$	(relation)

The mean is used extensively in:

- normalization layer computations,
- loss functions,
- statistical analysis,
- variance and standard deviation.

Implementation Principle

Compute:

$$\text{mean}(x) = \frac{1}{n}(x_1 + x_2 + \cdots + x_n)$$

Programming Task

Write a Python function that computes the mean of a vector.

```
def E10(x):
    s = 0
    for i in range(len(x)):
        s += x[i]
    return s / len(x)
```

34 T01 – E11 : 1D Variance Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n,$$

and let its mean be

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i.$$

The **variance** of the vector is the mapping

$$\text{Var} : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$$

defined by:

$$\text{Var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

Variance measures the spread of the data around the mean.

Properties:

$\text{Var}(x) \geq 0$	(non-negativity)
$\text{Var}(x) = 0 \iff x_1 = x_2 = \cdots = x_n$	(zero spread)
$\text{Var}(x + c) = \text{Var}(x)$	(translation invariance)
$\text{Var}(\alpha x) = \alpha^2 \text{Var}(x)$	(scale invariance)

Expanded form:

$$\text{Var}(x) = \frac{1}{n} [(x_1^2 + x_2^2 + \cdots + x_n^2) - n\mu^2]$$

Implementation Principle

Compute:

$$\mu = \text{mean}(x)$$

$$\text{Var}(x) = \frac{1}{n} \sum (x_i - \mu)^2$$

Programming Task

Write a Python function that computes the variance of a vector.

```
def E11(x):
    n = len(x)

    # compute mean
    mean = 0
    for i in range(n):
        mean += x[i]
    mean /= n

    # compute variance
    var = 0
    for i in range(n):
        diff = x[i] - mean
        var += diff * diff
    var /= n

    return var
```

35 T01 – E12 : 1D Standard Deviation Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n,$$

and let its variance be

$$\text{Var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2, \quad \mu = \frac{1}{n} \sum_{i=1}^n x_i.$$

The **standard deviation** of x is the mapping

$$\text{Std}(x) : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$$

defined by:

$$\text{Std}(x) = \sqrt{\text{Var}(x)}$$

Standard deviation measures the average amount of variation from the mean.
Properties:

$\text{Std}(x) \geq 0$	(non-negativity)
$\text{Std}(x) = 0 \iff x_i = \mu \ \forall i$	(zero spread)
$\text{Std}(x + c) = \text{Std}(x)$	(translation invariance)
$\text{Std}(\alpha x) = \alpha \text{Std}(x)$	(scaling)

Relationship to Variance:

$$\text{Std}(x)^2 = \text{Var}(x)$$

Implementation Principle

Compute:

$$\text{Std}(x) = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

Programming Task

Write a Python function that computes the standard deviation of a vector.

```
import math

def E12(x):
    n = len(x)

    # compute mean
    mean = 0
    for i in range(n):
        mean += x[i]
    mean /= n

    # compute variance
    var = 0
    for i in range(n):
        diff = x[i] - mean
        var += diff * diff
    var /= n

    # standard deviation
    std = math.sqrt(var)
    return std
```

36 T01 – E13 : 1D Cumulative Sum Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **cumulative sum** (prefix sum) is the mapping

$$\text{cumsum} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

defined by:

$$\text{cumsum}(x)_k = \sum_{i=1}^k x_i \quad \text{for } k = 1, 2, \dots, n$$

Thus the output vector is:

$$\text{cumsum}(x) = \left(x_1, x_1 + x_2, x_1 + x_2 + x_3, \dots, \sum_{i=1}^n x_i \right)$$

Properties:

$$\begin{aligned} \text{cumsum}(x)_k - \text{cumsum}(x)_{k-1} &= x_k \\ \text{cumsum}(x + y) &= \text{cumsum}(x) + \text{cumsum}(y) \\ \text{cumsum}(\alpha x) &= \alpha \text{ cumsum}(x) \end{aligned}$$

Implementation Principle

Compute partial sums:

$$s_k = s_{k-1} + x_k, \quad s_0 = 0$$

Programming Task

Write a Python function that computes the cumulative sum of a vector.

```
def E13(x):
    out = []
    s = 0
    for i in range(len(x)):
        s += x[i]
        out.append(s)
    return out
```

37 T01 – E14 : 1D Cumulative Product Law

Mathematical Definition

Let a vector

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n.$$

The **cumulative product** (prefix product) is the mapping

$$\text{cumprod} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

defined by:

$$\text{cumprod}(x)_k = \prod_{i=1}^k x_i \quad \text{for } k = 1, 2, \dots, n$$

Thus the output vector is:

$$\text{cumprod}(x) = \left(x_1, x_1 x_2, x_1 x_2 x_3, \dots, \prod_{i=1}^n x_i \right)$$

Properties:

$$\begin{aligned} \text{cumprod}(x)_k &= x_k \cdot \text{cumprod}(x)_{k-1} \\ \text{cumprod}(x \odot y) &= \text{cumprod}(x) \odot \text{cumprod}(y) \\ \text{cumprod}(\alpha x) &= \alpha^k \cdot \text{cumprod}(x)_k \end{aligned}$$

Implementation Principle

Compute partial products:

$$p_k = p_{k-1} \cdot x_k, \quad p_0 = 1$$

Programming Task

Write a Python function that computes the cumulative product of a vector.

```
def E14(x):
    out = []
    p = 1
    for i in range(len(x)):
        p *= x[i]
        out.append(p)
    return out
```

38 Matrix Laws (2D)

The purpose of **T02** is to introduce the fundamental laws of **2-dimensional matrix mathematics** using Python, PyTorch, and CUDA. This track focuses on the core matrix operations that form the backbone of all deep learning models, scientific computing, numerical analysis, and GPU-based tensor operations.

A matrix is represented as:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \in \mathbb{R}^{n \times m}$$

In **T02**, each exercise presents one matrix law. The student reads the mathematical rule, understands how the operation works, and then implements it using PyTorch–CUDA.

This track includes the following laws:

- E00 – 2D Matrix Multiplication Law
- E01 – 2D Hadamard Matrix Law
- E02 – 2D Frobenius Inner Product Law
- E03 – 2D Determinant Law
- E04 – 2D Inverse Matrix Law
- E05 – 2D Transpose Law
- E06 – 2D Trace Law
- E07 – 2D Cofactor Law
- E08 – 2D Adjugate Law
- E09 – 2D Rank Law
- E10 – 2D Row Sum Law
- E11 – 2D Column Sum Law
- E12 – 2D Row Mean Law
- E13 – 2D Column Mean Law
- E14 – 2D Broadcasting Law

By completing **T02**, the student will gain a strong and rigorous understanding of matrix algebra and will learn how to translate matrix laws into efficient PyTorch–CUDA operations. This knowledge is essential for learning higher-rank tensors (T03) and for building neural networks, attention mechanisms, and large-scale GPU computational systems.

39 T02 – E00 : 2D Matrix Multiplication Law

Mathematical Definition

Let two matrices

$$A \in \mathbb{R}^{n \times m}, \quad B \in \mathbb{R}^{m \times p}.$$

The **matrix multiplication** (also called matrix product) is the mapping

$$AB : \mathbb{R}^{n \times m} \times \mathbb{R}^{m \times p} \rightarrow \mathbb{R}^{n \times p}$$

defined by:

$$(AB)_{ij} = \sum_{k=1}^m A_{ik}B_{kj}$$

Thus each element of the output matrix is computed by:

$$AB = \begin{pmatrix} \sum_{k=1}^m A_{1k}B_{k1} & \dots & \sum_{k=1}^m A_{1k}B_{kp} \\ \sum_{k=1}^m A_{2k}B_{k1} & \dots & \sum_{k=1}^m A_{2k}B_{kp} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^m A_{nk}B_{k1} & \dots & \sum_{k=1}^m A_{nk}B_{kp} \end{pmatrix}$$

Compatibility requirement:

$$A \in \mathbb{R}^{n \times m}, \quad B \in \mathbb{R}^{m \times p}$$

The middle dimensions must match.

Properties:

$(AB)C = A(BC)$	(associativity)
$A(B + C) = AB + AC$	(left distributivity)
$(A + B)C = AC + BC$	(right distributivity)
$IA = A$	(identity matrix)
$AB \neq BA$	(non-commutative in general)

Implementation Principle

Compute each entry as:

$$z_{ij} = \sum_{k=1}^m A_{ik}B_{kj}$$

Programming Task

Write a Python function that multiplies two matrices.

```
def E00(A, B):
    # A is n x m
    # B is m x p
    n = len(A)
    m = len(A[0])
    p = len(B[0])

    # output matrix: n x p
    Z = [[0 for _ in range(p)] for _ in range(n)]

    for i in range(n):
        for j in range(p):
            s = 0
            for k in range(m):
                s += A[i][k] * B[k][j]
            Z[i][j] = s

    return Z
```

40 T02 – E01 : 2D Hadamard Matrix Law

Mathematical Definition

Let two matrices of identical shape

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}, \quad B = (B_{ij}) \in \mathbb{R}^{n \times m}.$$

The **Hadamard product** (element-wise multiplication) is the mapping

$$A \odot B : \mathbb{R}^{n \times m} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$$

defined by:

$$(A \odot B)_{ij} = A_{ij} B_{ij}$$

Thus:

$$A \odot B = \begin{pmatrix} A_{11}B_{11} & \dots & A_{1m}B_{1m} \\ A_{21}B_{21} & \dots & A_{2m}B_{2m} \\ \vdots & \ddots & \vdots \\ A_{n1}B_{n1} & \dots & A_{nm}B_{nm} \end{pmatrix}$$

Shape requirement:

$$A, B \in \mathbb{R}^{n \times m}$$

Properties:

$A \odot B = B \odot A$	(commutative)
$A \odot (B + C) = (A \odot B) + (A \odot C)$	(distributive)
$(\alpha A) \odot B = \alpha(A \odot B)$	(scalar associativity)
$A \odot I = \text{diag}(A_{11}, A_{22}, \dots)$	(identity pattern)

Implementation Principle

Element-wise:

$$Z_{ij} = A_{ij} \cdot B_{ij}$$

Programming Task

Write a Python function that computes the Hadamard product of two matrices.

```
def E01(A, B):
    n = len(A)
    m = len(A[0])

    Z = [[0 for _ in range(m)] for _ in range(n)]

    for i in range(n):
        for j in range(m):
            Z[i][j] = A[i][j] * B[i][j]

    return Z
```

41 T02 – E02 : 2D Frobenius Inner Product Law

Mathematical Definition

Let two matrices of the same shape

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}, \quad B = (B_{ij}) \in \mathbb{R}^{n \times m}.$$

The **Frobenius inner product** is the mapping:

$$\langle A, B \rangle_F : \mathbb{R}^{n \times m} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}$$

defined by:

$$\langle A, B \rangle_F = \sum_{i=1}^n \sum_{j=1}^m A_{ij} B_{ij}$$

This is equivalent to flattening the matrices into vectors and applying the standard dot product.

Equivalent forms:

$$\langle A, B \rangle_F = \text{trace}(A^\top B)$$

$$\langle A, B \rangle_F = (A \odot B) \text{ summed element-wise}$$

Properties:

$\langle A, B \rangle_F = \langle B, A \rangle_F$	(symmetry)
$\langle A + C, B \rangle_F = \langle A, B \rangle_F + \langle C, B \rangle_F$	(linearity)
$\langle \alpha A, B \rangle_F = \alpha \langle A, B \rangle_F$	(scalar associativity)
$\langle A, A \rangle_F = \ A\ _F^2$	(relation to Frobenius norm)

Implementation Principle

Compute:

$$s = \sum_{i=1}^n \sum_{j=1}^m A_{ij} B_{ij}$$

Programming Task

Write a Python function that computes the Frobenius inner product of two matrices.

```
def E02(A, B):
    n = len(A)
    m = len(A[0])
    s = 0

    for i in range(n):
        for j in range(m):
            s += A[i][j] * B[i][j]

    return s
```

42 T02 – E03 : 2D Determinant Law

Mathematical Definition

The determinant is defined only for square matrices.

Let

$$A = (A_{ij}) \in \mathbb{R}^{n \times n}.$$

The **determinant** is the mapping

$$\det : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$$

For a 2×2 matrix:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

the determinant is:

$$\det(A) = ad - bc$$

For a 3×3 matrix:

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

$$\det(A) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

General (Leibniz formula):

$$\det(A) = \sum_{\sigma \in S_n} \text{sgn}(\sigma) \prod_{i=1}^n A_{i,\sigma(i)}$$

Geometric interpretation:

- $|\det(A)|$ is the volume scaling factor of the linear transformation.
- $\det(A) = 0$ means the matrix is singular (not invertible).

Properties:

$$\begin{aligned}\det(AB) &= \det(A)\det(B) \\ \det(A^\top) &= \det(A) \\ \det(\alpha A) &= \alpha^n \det(A) \\ A \text{ invertible} &\iff \det(A) \neq 0\end{aligned}$$

Implementation Principle

We implement only the 2×2 case (sufficient for fundamental laws).

$$\det(A) = A_{11}A_{22} - A_{12}A_{21}$$

Programming Task

Write a Python function that computes the determinant of a 2×2 matrix.

```
def E03(A):
    # A is 2x2
    a = A[0][0]
    b = A[0][1]
    c = A[1][0]
    d = A[1][1]
    return a * d - b * c
```

43 T02 – E04 : 2D Inverse Matrix Law

Mathematical Definition

The matrix inverse is defined only for square, non-singular matrices.

Let

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathbb{R}^{2 \times 2}, \quad \det(A) = ad - bc \neq 0.$$

The **inverse** of A is the matrix

$$A^{-1} \in \mathbb{R}^{2 \times 2}$$

defined by:

$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

This satisfies:

$$AA^{-1} = A^{-1}A = I_2$$

General properties:

$$\begin{aligned}(AB)^{-1} &= B^{-1}A^{-1} \\ (A^\top)^{-1} &= (A^{-1})^\top \\ (\alpha A)^{-1} &= \frac{1}{\alpha}A^{-1}\end{aligned}$$

A matrix is invertible if and only if:

$$\det(A) \neq 0$$

Implementation Principle

Compute:

$$A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Programming Task

Write a Python function that computes the inverse of a 2×2 matrix.

```
def E04(A):
    # A is 2x2
    a = A[0][0]
    b = A[0][1]
    c = A[1][0]
    d = A[1][1]

    det = a * d - b * c
    if det == 0:
        raise ValueError("Matrix is singular")

    inv_det = 1 / det

    return [
        [d * inv_det, -b * inv_det],
        [-c * inv_det, a * inv_det]
    ]
```

44 T02 – E05 : 2D Transpose Law

Mathematical Definition

Let a matrix

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}.$$

The **transpose** of A is the matrix

$$A^\top \in \mathbb{R}^{m \times n}$$

defined by swapping rows and columns:

$$(A^\top)_{ij} = A_{ji}$$

Thus:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \implies A^\top = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m} & a_{2m} & \dots & a_{nm} \end{pmatrix}$$

Key properties:

$$\begin{aligned} (A^\top)^\top &= A \\ (A + B)^\top &= A^\top + B^\top \\ (\alpha A)^\top &= \alpha A^\top \\ (AB)^\top &= B^\top A^\top \end{aligned}$$

The transpose is one of the most fundamental matrix operations used in:

- linear algebra,
- neural network layers,
- attention mechanisms,
- gradient computation.

Implementation Principle

Compute:

$$(A^\top)_{ij} = A_{ji}$$

Programming Task

Write a Python function that computes the transpose of a matrix.

```
def E05(A):
    n = len(A)      # rows
    m = len(A[0])   # columns

    # output matrix: m x n
    T = [[0 for _ in range(n)] for _ in range(m)]

    for i in range(n):
        for j in range(m):
            T[j][i] = A[i][j]

    return T
```

45 T02 – E06 : 2D Trace Law

Mathematical Definition

The trace is defined only for square matrices.

Let

$$A = (A_{ij}) \in \mathbb{R}^{n \times n}.$$

The **trace** of A is the mapping

$$\text{tr} : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$$

defined by the sum of the diagonal elements:

$$\text{tr}(A) = \sum_{i=1}^n A_{ii}$$

Example:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \Rightarrow \text{tr}(A) = a + d$$

Properties:

$$\begin{aligned} \text{tr}(A + B) &= \text{tr}(A) + \text{tr}(B) \\ \text{tr}(\alpha A) &= \alpha \text{tr}(A) \\ \text{tr}(AB) &= \text{tr}(BA) \\ \text{tr}(A^\top) &= \text{tr}(A) \end{aligned}$$

Geometric/linear interpretation:

$$\text{tr}(A) = \text{sum of eigenvalues of } A$$

Implementation Principle

Compute:

$$\text{tr}(A) = A_{11} + A_{22} + \dots + A_{nn}$$

Programming Task

Write a Python function that computes the trace of a square matrix.

```
def E06(A):
    n = len(A)    # A is n x n
    s = 0
    for i in range(n):
        s += A[i][i]
    return s
```

46 T02 – E07 : 2D Cofactor Law

Mathematical Definition

Let

$$A = (A_{ij}) \in \mathbb{R}^{n \times n}$$

be a square matrix.

The **cofactor** of element A_{ij} is defined as:

$$C_{ij} = (-1)^{i+j} M_{ij}$$

where M_{ij} is the **minor** of A_{ij} , defined as:

$$M_{ij} = \det(A_{(i,j)})$$

and $A_{(i,j)}$ is the matrix obtained by deleting row i and column j .

For a 2×2 matrix:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

the cofactor matrix is:

$$C = \begin{pmatrix} (+1)d & (-1)c \\ (-1)b & (+1)a \end{pmatrix} = \begin{pmatrix} d & -c \\ -b & a \end{pmatrix}$$

This matches the formula:

$$C = \begin{pmatrix} A_{22} & -A_{12} \\ -A_{21} & A_{11} \end{pmatrix}$$

Properties:

$$\begin{aligned} \text{adj}(A) &= C^\top \\ A^{-1} &= \frac{1}{\det(A)} \text{adj}(A) \\ \det(A) &= \sum_{j=1}^n A_{ij} C_{ij} \end{aligned}$$

Implementation Principle

For 2×2 :

$$C = \begin{pmatrix} d & -c \\ -b & a \end{pmatrix}$$

Programming Task

Write a Python function that computes the cofactor matrix of a 2×2 matrix.

```
def E07(A):
    # A is 2x2
    a = A[0][0]
    b = A[0][1]
    c = A[1][0]
    d = A[1][1]

    return [
        [d, -c],
        [-b, a]
    ]
```

47 T02 – E08 : 2D Adjugate Law

Mathematical Definition

Let a square matrix

$$A \in \mathbb{R}^{n \times n}.$$

The **adjugate matrix** (also called the classical adjoint) is defined as:

$$\text{adj}(A) = C^\top$$

where C is the cofactor matrix of A .

For the 2×2 case:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

the cofactor matrix is:

$$C = \begin{pmatrix} d & -c \\ -b & a \end{pmatrix}$$

Thus the adjugate is:

$$\text{adj}(A) = C^\top = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Key relation:

$$A^{-1} = \frac{1}{\det(A)} \text{adj}(A) \quad \text{for } \det(A) \neq 0$$

Properties:

$$\text{adj}(AB) = \text{adj}(B) \text{adj}(A)$$

$$\text{adj}(A^\top) = (\text{adj}(A))^\top$$

$$A \text{adj}(A) = \det(A)I_n$$

Implementation Principle

For a 2×2 matrix:

$$\text{adj}(A) = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Programming Task

Write a Python function that computes the adjugate of a 2×2 matrix.

```
def E08(A):
    # A is 2x2
    a = A[0][0]
    b = A[0][1]
    c = A[1][0]
    d = A[1][1]

    return [
        [d, -b],
        [-c, a]
    ]
```

48 T02 – E09 : 2D Rank Law

Mathematical Definition

Let

$$A \in \mathbb{R}^{n \times m}.$$

The **rank** of a matrix is the mapping:

$$\text{rank} : \mathbb{R}^{n \times m} \rightarrow \mathbb{N}$$

defined as the dimension of the vector space spanned by its rows or columns.

Equivalent definitions:

- Number of linearly independent rows.
- Number of linearly independent columns.
- Number of nonzero singular values of A .
- The largest k such that A contains a nonzero $k \times k$ minor.

For a 2×2 matrix:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$
$$\text{rank}(A) = \begin{cases} 2, & \text{if } ad - bc \neq 0 \\ 1, & \text{if } (a, b) \parallel (c, d) \\ 0, & A = 0 \end{cases}$$

Key properties:

$$\begin{aligned} \text{rank}(A) &= \text{rank}(A^\top) \\ \text{rank}(AB) &\leq \min(\text{rank}(A), \text{rank}(B)) \\ A \text{ invertible} &\iff \text{rank}(A) = n \end{aligned}$$

Implementation Principle

For a 2×2 :

$$\begin{aligned} \det(A) &= ad - bc \\ \text{rank}(A) &= \begin{cases} 2, & \det(A) \neq 0 \\ 1, & \text{any row/column nonzero} \\ 0, & \text{all entries zero} \end{cases} \end{aligned}$$

Programming Task

Write a Python function that computes the rank of a 2×2 matrix.

```
def E09(A):  
    a = A[0][0]  
    b = A[0][1]  
    c = A[1][0]  
    d = A[1][1]  
  
    det = a * d - b * c  
  
    if det != 0:  
        return 2 # full rank  
  
    # check if any row or column is nonzero  
    if (a != 0 or b != 0) or (c != 0 or d != 0):  
        return 1  
  
    return 0 # zero matrix
```

49 T02 – E10 : 2D Row Sum Law

Mathematical Definition

Let a matrix

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}.$$

The **row sum** operation maps each row of A to the sum of its elements:

$$\text{rowsum}(A) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$$

defined by:

$$\text{rowsum}(A)_i = \sum_{j=1}^m A_{ij}$$

Thus the output is a column vector:

$$\text{rowsum}(A) = \begin{pmatrix} \sum_{j=1}^m A_{1j} \\ \sum_{j=1}^m A_{2j} \\ \vdots \\ \sum_{j=1}^m A_{nj} \end{pmatrix}$$

Properties:

$$\begin{aligned} \text{rowsum}(A + B) &= \text{rowsum}(A) + \text{rowsum}(B) \\ \text{rowsum}(\alpha A) &= \alpha \cdot \text{rowsum}(A) \\ \text{rowsum}(A^\top) &= \text{colsum}(A) \end{aligned}$$

Implementation Principle

Compute each row's sum:

$$s_i = \sum_{j=1}^m A_{ij}$$

Programming Task

Write a Python function that computes the sum of each row of a matrix.

```
def E10(A):
    n = len(A)
    m = len(A[0])

    out = [0 for _ in range(n)]

    for i in range(n):
        s = 0
        for j in range(m):
            s += A[i][j]
        out[i] = s

    return out
```

50 T02 – E11 : 2D Column Sum Law

Matsectionical Definition

Let a matrix

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}.$$

The **column sum** operation maps each column of A to the sum of its elements:

$$\text{colsum}(A) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$$

defined by:

$$\text{colsum}(A)_j = \sum_{i=1}^n A_{ij}$$

Thus the output vector is:

$$\text{colsum}(A) = \begin{pmatrix} \sum_{i=1}^n A_{i1} \\ \sum_{i=1}^n A_{i2} \\ \vdots \\ \sum_{i=1}^n A_{im} \end{pmatrix}$$

Properties:

$$\begin{aligned} \text{colsum}(A + B) &= \text{colsum}(A) + \text{colsum}(B) \\ \text{colsum}(\alpha A) &= \alpha \cdot \text{colsum}(A) \\ \text{colsum}(A^\top) &= \text{rowsum}(A) \end{aligned}$$

Implementation Principle

Compute each column's sum:

$$s_j = \sum_{i=1}^n A_{ij}$$

Programming Task

Write a Python function that computes the sum of each column of a matrix.

```
def E11(A):
    n = len(A)
    m = len(A[0])

    out = [0 for _ in range(m)]

    for j in range(m):
        s = 0
        for i in range(n):
            s += A[i][j]
        out[j] = s

    return out
```

51 T02 – E12 : 2D Row Mean Law

Mathematical Definition

Let a matrix

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}.$$

The **row mean** computes the average of each row:

$$\text{rowmean}(A) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$$

defined by:

$$\text{rowmean}(A)_i = \frac{1}{m} \sum_{j=1}^m A_{ij}$$

Thus the output vector is:

$$\text{rowmean}(A) = \begin{pmatrix} \frac{1}{m} \sum_{j=1}^m A_{1j} \\ \frac{1}{m} \sum_{j=1}^m A_{2j} \\ \vdots \\ \frac{1}{m} \sum_{j=1}^m A_{nj} \end{pmatrix}$$

Properties:

$$\begin{aligned} \text{rowmean}(A + B) &= \text{rowmean}(A) + \text{rowmean}(B) \\ \text{rowmean}(\alpha A) &= \alpha \cdot \text{rowmean}(A) \\ \text{rowmean}(A^\top) &= \text{colmean}(A) \end{aligned}$$

Implementation Principle

Compute the mean of each row:

$$\text{rowmean}(A)_i = \frac{1}{m} \sum_{j=1}^m A_{ij}$$

Programming Task

Write a Python function that computes the mean of each row.

```
def E12(A):
    n = len(A)
    m = len(A[0])

    out = [0 for _ in range(n)]

    for i in range(n):
        s = 0
        for j in range(m):
            s += A[i][j]
        out[i] = s / m

    return out
```

52 T02 – E13 : 2D Column Mean Law

Mathematical Definition

Let a matrix

$$A = (A_{ij}) \in \mathbb{R}^{n \times m}.$$

The **column mean** computes the average of each column:

$$\text{colmean}(A) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$$

defined by:

$$\text{colmean}(A)_j = \frac{1}{n} \sum_{i=1}^n A_{ij}$$

Thus the output vector is:

$$\text{colmean}(A) = \begin{pmatrix} \frac{1}{n} \sum_{i=1}^n A_{i1} \\ \frac{1}{n} \sum_{i=1}^n A_{i2} \\ \vdots \\ \frac{1}{n} \sum_{i=1}^n A_{im} \end{pmatrix}$$

Properties:

$$\begin{aligned}\text{colmean}(A + B) &= \text{colmean}(A) + \text{colmean}(B) \\ \text{colmean}(\alpha A) &= \alpha \cdot \text{colmean}(A) \\ \text{colmean}(A^\top) &= \text{rowmean}(A)\end{aligned}$$

Implementation Principle

Compute the mean of each column:

$$\text{colmean}(A)_j = \frac{1}{n} \sum_{i=1}^n A_{ij}$$

Programming Task

Write a Python function that computes the mean of each column.

```
def E13(A):
    n = len(A)
    m = len(A[0])

    out = [0 for _ in range(m)]

    for j in range(m):
        s = 0
        for i in range(n):
            s += A[i][j]
        out[j] = s / n

    return out
```

53 T02 – E14 : 2D Broadcasting Law

Mathematical Definition

Broadcasting is a rule that allows operations between arrays of different shapes by automatically expanding dimensions when mathematically valid.

Let

$$A \in \mathbb{R}^{n \times m}, \quad b \in \mathbb{R}^m.$$

The broadcasting rule allows:

$$A + b$$

by expanding b into:

$$b \rightarrow \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ b_1 & b_2 & \dots & b_m \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \dots & b_m \end{pmatrix} \in \mathbb{R}^{n \times m}$$

General broadcasting rule:

Two shapes (d_1, d_2, \dots, d_k) and (e_1, e_2, \dots, e_k) are compatible if for every dimension:

$$d_i = e_i \quad \text{or} \quad d_i = 1 \quad \text{or} \quad e_i = 1$$

Examples:

$$(n, m) + (m) \quad \text{valid}$$

$$(n, m) + (n, 1) \quad \text{valid}$$

$$(n, m) + (n, m) \quad \text{valid}$$

$$(n, m) + (n) \quad \text{invalid}$$

Implementation Principle

When computing:

$$Z_{ij} = A_{ij} + b_j$$

Broadcast:

b_j repeated for each row

Programming Task

Write a Python function that performs broadcasting addition between a matrix and a vector.

```
def E14(A, b):
    n = len(A)
    m = len(A[0])

    # b must be length m
    if len(b) != m:
        raise ValueError("Broadcast shape mismatch")

    Z = [[0 for _ in range(m)] for _ in range(n)]

    for i in range(n):
        for j in range(m):
            Z[i][j] = A[i][j] + b[j]

    return Z
```

54 Higher-Dimension Tensor Laws

The purpose of **T03** is to introduce the mathematical laws of **higher-dimensional tensors** (3D and above) using Python, PyTorch, and CUDA. This track extends the concepts learned in T00 (tensor algebra), T01 (vector laws), and T02 (matrix laws), and moves into the realm of **multi-dimensional** tensor operations that are essential in deep learning, scientific computing, computer vision, transformer models, and GPU tensor cores.

A 3D tensor is written as:

$$\mathcal{T} \in \mathbb{R}^{n \times m \times p}$$

and higher-order tensors generalize this structure to 4D, 5D, and beyond.

In **T03**, each exercise presents one higher-dimensional tensor law. The student reads the mathematical definition, understands the operation, and then implements it using optimized PyTorch–CUDA code.

This track includes the following laws:

- E00 – 3D Tensor Multiplication Law
- E01 – 3D Outer Product Law
- E02 – 3D Einstein Summation Law
- E03 – Batch Matrix Multiplication Law
- E04 – Tensor Contraction Law
- E05 – Tensor Repetition Law
- E06 – Tensor Tiling Law
- E07 – Tensor Broadcasting Law
- E08 – Tensor Indexing Law
- E09 – Tensor Advanced Slicing Law

- E10 – Tensor Padding Law
- E11 – Tensor Cropping Law
- E12 – Tensor Stacking Law
- E13 – Tensor Splitting Law
- E14 – Tensor Chunking Law

By completing **T03**, the student will move from basic tensor algebra to full mastery of multi-dimensional tensor operations. These skills are fundamental for understanding CNNs, transformers, attention mechanisms, automatic differentiation systems, and GPU-accelerated tensor programs.

55 T03 – E00 : 3D Tensor Multiplication Law

Mathematical Definition

Let a 3D tensor

$$\mathcal{A} \in \mathbb{R}^{n \times m \times p}$$

and a matrix

$$B \in \mathbb{R}^{p \times q}.$$

The **3D tensor–matrix multiplication** is defined by multiplying each matrix slice of the tensor along the last dimension by B :

$$\mathcal{C}_{i,j,k} = \sum_{t=1}^p \mathcal{A}_{i,j,t} B_{t,k}$$

Thus the resulting tensor has shape:

$$\mathcal{C} \in \mathbb{R}^{n \times m \times q}$$

Slice interpretation:

$$\mathcal{A}_{i,:,:} \in \mathbb{R}^{m \times p} \Rightarrow \mathcal{C}_{i,:,:} = \mathcal{A}_{i,:,:} \cdot B$$

Properties:

$$\begin{aligned} (\mathcal{A}\mathcal{B})\mathcal{C} &= \mathcal{A}(\mathcal{B}\mathcal{C}) \\ (\alpha\mathcal{A})\mathcal{B} &= \alpha(\mathcal{A}\mathcal{B}) \\ (\mathcal{A} + \mathcal{D})\mathcal{B} &= \mathcal{A}\mathcal{B} + \mathcal{D}\mathcal{B} \end{aligned}$$

This law generalizes normal matrix multiplication to an entire batch of matrices.

Implementation Principle

For each $i \in \{1, \dots, n\}$:

$$\mathcal{C}_{i,:,:} = \mathcal{A}_{i,:,:} \cdot B$$

Programming Task

Write a Python function that multiplies a 3D tensor with a matrix.

```
def E00(A, B):
    # A shape: n x m x p
    # B shape: p x q
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])
    q = len(B[0])
```

```

# Output tensor: n x m x q
C = [[[0 for _ in range(q)] for _ in range(m)] for _ in range(n)]

for i in range(n):
    for r in range(m):
        for c in range(q):
            s = 0
            for t in range(p):
                s += A[i][r][t] * B[t][c]
            C[i][r][c] = s

return C

```

56 T03 – E01 : 3D Outer Product Law

Mathematical Definition

Let two vectors:

$$u \in \mathbb{R}^m, \quad v \in \mathbb{R}^n, \quad w \in \mathbb{R}^p.$$

The **3D outer product** constructs a rank-3 tensor:

$$\mathcal{T} = u \otimes v \otimes w \in \mathbb{R}^{m \times n \times p}$$

defined by:

$$\mathcal{T}_{i,j,k} = u_i v_j w_k$$

This is a direct generalization of the 2D outer product:

$$u \otimes v = (u_i v_j)$$

to 3D:

$$(u \otimes v \otimes w)_{i,j,k} = u_i \cdot v_j \cdot w_k$$

Tensor interpretation:

- Each fixed k slice is the 2D outer product $u \otimes v$ scaled by w_k .
- The resulting tensor has axes corresponding to the dimensions of (u, v, w) .

Properties:

$$\begin{aligned} \alpha(u \otimes v \otimes w) &= (\alpha u) \otimes v \otimes w \\ u \otimes (v + t) \otimes w &= (u \otimes v \otimes w) + (u \otimes t \otimes w) \\ \text{rank}(\mathcal{T}) &= 1 \text{ (for nonzero vectors)} \end{aligned}$$

This operation is fundamental in:

- tensor decomposition,
- attention mechanisms,
- multi-dimensional feature interactions,
- higher-order linear algebra.

Implementation Principle

Compute:

$$\mathcal{T}_{i,j,k} = u_i v_j w_k$$

Programming Task

Write a Python function that computes the 3D outer product.

```
def E01(u, v, w):
    m = len(u)
    n = len(v)
    p = len(w)

    T = [[[0 for _ in range(p)] for _ in range(n)] for _ in range(m)]

    for i in range(m):
        for j in range(n):
            for k in range(p):
                T[i][j][k] = u[i] * v[j] * w[k]

    return T
```

57 T03 – E02 : 3D Einstein Summation Law

Mathematical Definition

Einstein summation (also called `einsum`) is a compact notation for tensor operations where repeated indices are automatically summed.

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}$$

and a matrix:

$$B = B_{kt} \in \mathbb{R}^{p \times q}.$$

The Einstein summation rule:

$$\mathcal{C}_{ijt} = \sum_{k=1}^p \mathcal{A}_{ijk} B_{kt}$$

is written simply as:

$$\mathcal{C} = \text{einsum}(“ijk, kt \rightarrow ijt”, \mathcal{A}, B)$$

General rule:

Whenever an index appears twice:

$$X_{abcd} Y_{cde} \Rightarrow \sum_{c,d}$$

This allows extremely flexible tensor operations such as:

$$\text{einsum}(“abc, acd \rightarrow bd”)$$

Properties:

- Automatically handles summation over repeated indices.
- Can express matrix multiplication, outer products, contractions.
- More general than broadcasting or matmul.

Examples of Einstein Patterns

$$\text{einsum}(“ij, jk \rightarrow ik”) = AB \quad (\text{matrix multiplication})$$

$$\text{einsum}(“i, j, k \rightarrow ijk”) = u \otimes v \otimes w$$

$$\text{einsum}(“ijk, ijk \rightarrow ”) = \langle \mathcal{A}, \mathcal{B} \rangle_F$$

Implementation Principle

Manually computing:

$$\mathcal{C}_{ijt} = \sum_{k=1}^p \mathcal{A}_{ijk} B_{kt}$$

Programming Task

Write a Python function that performs the 3D Einstein summation defined above.

```
def E02(A, B):
    # A shape: n x m x p
    # B shape: p x q
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])
    q = len(B[0])

    C = [[[0 for _ in range(q)] for _ in range(m)] for _ in range(n)]

    for i in range(n):
        for j in range(m):
            for t in range(q):
                s = 0
                for k in range(p):
                    s += A[i][j][k] * B[k][t]
                C[i][j][t] = s

    return C
```

58 T03 – E03 : Batch Matrix Multiplication Law

Mathematical Definition

Let a batch of matrices:

$$\mathcal{A} \in \mathbb{R}^{b \times n \times m}$$

and another batch:

$$\mathcal{B} \in \mathbb{R}^{b \times m \times p}.$$

The **batch matrix multiplication** computes:

$$\mathcal{C} = \mathcal{A} \times \mathcal{B} \quad \in \quad \mathbb{R}^{b \times n \times p}$$

Each batch slice multiplies independently:

$$\mathcal{C}_{t,i,j} = \sum_{k=1}^m \mathcal{A}_{t,i,k} \mathcal{B}_{t,k,j} \quad \text{for } t = 1, \dots, b$$

Thus:

$$\mathcal{C}_{t,:,:} = \mathcal{A}_{t,:,:} \cdot \mathcal{B}_{t,:,:}$$

This is the mathematical foundation of:

- multi-head attention,
- transformer blocks,
- CNN layers,
- batched GPU operations.

Shape Rule

$$(b, n, m) \times (b, m, p) \longrightarrow (b, n, p)$$

Properties

$$\begin{aligned} (\mathcal{A} + \mathcal{D})\mathcal{B} &= \mathcal{A}\mathcal{B} + \mathcal{D}\mathcal{B} \\ \mathcal{A}(\mathcal{B} + \mathcal{E}) &= \mathcal{A}\mathcal{B} + \mathcal{A}\mathcal{E} \\ (\alpha\mathcal{A})\mathcal{B} &= \alpha(\mathcal{A}\mathcal{B}) \end{aligned}$$

Each batch index is independent:

$$t \neq s \Rightarrow \mathcal{A}_{t,:,:}, \mathcal{B}_{s,:,:} \text{ is never mixed}$$

Implementation Principle

For each batch t :

$$\mathcal{C}_{t,i,j} = \sum_{k=1}^m \mathcal{A}_{t,i,k} \mathcal{B}_{t,k,j}$$

Programming Task

Write a Python function that performs batch matrix multiplication.

```
def E03(A, B):
    # A shape: b x n x m
    # B shape: b x m x p
    b = len(A)
    n = len(A[0])
    m = len(A[0][0])
    p = len(B[0][0])

    C = [[[0 for _ in range(p)] for _ in range(n)] for _ in range(b)]

    for t in range(b):
        for i in range(n):
            for j in range(p):
                s = 0
                for k in range(m):
                    s += A[t][i][k] * B[t][k][j]
                C[t][i][j] = s

    return C
```

59 T03 – E04 : Tensor Contraction Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}$$

and a 3D tensor:

$$\mathcal{B} = \mathcal{B}_{kjt} \in \mathbb{R}^{p \times m \times q}.$$

A **tensor contraction** reduces the dimensionality by summing over one or more shared indices. Here, we contract over indices j and k :

$$\mathcal{C}_{i,t} = \sum_{j=1}^m \sum_{k=1}^p \mathcal{A}_{ijk} \mathcal{B}_{kjt}$$

Resulting shape:

$$\mathcal{C} \in \mathbb{R}^{n \times q}$$

General definition:

If an index appears once in each tensor, it becomes a **summation index**. If an index appears only once overall, it becomes an **output index**.

Tensor contraction is a generalization of:

- matrix multiplication,
- dot product,
- trace,
- Einstein summation.

Einstein Summation Form

$$\mathcal{C}_{i,t} = \text{einsum}(”ijk, kjt \rightarrow it”, \mathcal{A}, \mathcal{B})$$

Examples

Dot product:

$$u_i v_i = \text{einsum}(”i, i \rightarrow ”)$$

Matrix multiplication:

$$A_{ik} B_{kj} = \text{einsum}(”ik, kj \rightarrow ij”)$$

Trace:

$$A_{ii} = \text{einsum}(”ii \rightarrow ”)$$

Implementation Principle

Compute:

$$\mathcal{C}_{i,t} = \sum_{j=1}^m \sum_{k=1}^p \mathcal{A}_{ijk} \mathcal{B}_{kjt}$$

Programming Task

Write a Python function that performs this contraction.

```
def E04(A, B):
    # A shape: n x m x p
    # B shape: p x m x q
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])
    q = len(B[0][0])

    C = [[0 for _ in range(q)] for _ in range(n)]

    for i in range(n):
        for t in range(q):
            s = 0
            for j in range(m):
                for k in range(p):
                    s += A[i][j][k] * B[k][j][t]
            C[i][t] = s

    return C
```

60 T03 – E05 : Tensor Repetition Law

Mathematical Definition

Let a tensor

$$\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_k}.$$

The **tensor repetition** operation repeats the tensor along a specified dimension.

For example, repeating a 3D tensor along the first dimension:

$$\text{repeat}(\mathcal{A}, r)_{i,j,k} = \mathcal{A}_{(i \bmod n_1), j, k}$$

The resulting shape is:

$$(r \cdot n_1) \times n_2 \times n_3$$

For each new index:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod n_1, j, k}$$

This generalizes to all dimensions.

Examples:

$$\text{repeat}(u, r) \in \mathbb{R}^{r \cdot m} \quad (\text{vector})$$

$$\text{repeat}(A, r) \in \mathbb{R}^{r \cdot n \times m} \quad (\text{matrix})$$

$$\text{repeat}(\mathcal{A}, r) \in \mathbb{R}^{r \cdot n_1 \times n_2 \times n_3} \quad (\text{3D tensor})$$

Properties

$$\text{repeat}(\alpha \mathcal{A}, r) = \alpha \cdot \text{repeat}(\mathcal{A}, r)$$

$$\text{repeat}(\mathcal{A} + \mathcal{B}, r) = \text{repeat}(\mathcal{A}, r) + \text{repeat}(\mathcal{B}, r)$$

$$\text{shape repeat rule: } (n_1, n_2, n_3) \rightarrow (rn_1, n_2, n_3)$$

Implementation Principle

For a 3D tensor:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod n}$$

Programming Task

Write a Python function that repeats a 3D tensor along the first dimension.

```
def E05(A, r):
    # A shape: n x m x p
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])

    # Output shape: (r*n) x m x p
    B = [[[0 for _ in range(p)] for _ in range(m)] for _ in range(r * n)]

    for i in range(r * n):
        src = i % n
        for j in range(m):
            for k in range(p):
                B[i][j][k] = A[src][j][k]

    return B
```

61 T03 – E06 : Tensor Tiling Law

Mathematical Definition

Let a tensor

$$\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_k}.$$

The **tensor tiling** operation repeats the entire tensor along multiple dimensions, producing a larger tensor composed of repeated blocks of the original.

For a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{n \times m \times p}$$

and tile factors:

$$(r_1, r_2, r_3) \in \mathbb{N}^3,$$

the tiled tensor is:

$$\mathcal{B} = \text{tile}(\mathcal{A}, (r_1, r_2, r_3)) \in \mathbb{R}^{(r_1 n) \times (r_2 m) \times (r_3 p)}$$

Definition:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod n, j \bmod m, k \bmod p}$$

This creates a grid of repeated blocks of \mathcal{A} .

Example (2D case):

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \text{tile}(A, (2, 2)) = \begin{pmatrix} a & b & a & b \\ c & d & c & d \\ a & b & a & b \\ c & d & c & d \end{pmatrix}$$

Properties

$$\begin{aligned} \text{tile}(\alpha \mathcal{A}, r) &= \alpha \text{ tile}(\mathcal{A}, r) \\ \text{tile}(\mathcal{A} + \mathcal{B}, r) &= \text{tile}(\mathcal{A}, r) + \text{tile}(\mathcal{B}, r) \end{aligned}$$

Shape:

$$(n, m, p) \rightarrow (r_1 n, r_2 m, r_3 p)$$

Implementation Principle

Compute:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod n, j \bmod m, k \bmod p}$$

Programming Task

Write a Python function that tiles a 3D tensor along all dimensions.

```
def E06(A, r1, r2, r3):
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])

    B = [[[0 for _ in range(r3 * p)]
          for _ in range(r2 * m)]
          for _ in range(r1 * n)]

    for i in range(r1 * n):
        for j in range(r2 * m):
            for k in range(r3 * p):
                B[i][j][k] = A[i % n][j % m][k % p]

    return B
```

62 T03 – E07 : Tensor Broadcasting Law

Mathematical Definition

Let tensors

$$\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_k}, \quad \mathcal{B} \in \mathbb{R}^{m_1 \times m_2 \times \dots \times m_k}.$$

Broadcasting allows element-wise operations between tensors of different shapes by expanding dimensions according to a compatibility rule.

Two dimensions n_i and m_i are **broadcast compatible** if:

$$n_i = m_i \quad \text{or} \quad n_i = 1 \quad \text{or} \quad m_i = 1.$$

During broadcasting:

if a dimension is 1, it is repeated to match the other tensor.

Example:

$$(3, 1, 5) \text{ and } (1, 4, 5)$$

broadcast to:

$$(3, 4, 5)$$

3D Broadcasting Example

Let:

$$\mathcal{A} \in \mathbb{R}^{n \times 1 \times p}, \quad \mathcal{B} \in \mathbb{R}^{1 \times m \times p}.$$

Broadcast result:

$$\mathcal{C} = \mathcal{A} \oplus \mathcal{B} \in \mathbb{R}^{n \times m \times p}$$

Element-wise:

$$\mathcal{C}_{i,j,k} = \mathcal{A}_{i,1,k} + \mathcal{B}_{1,j,k}$$

Properties

$$\mathcal{A} \oplus (\mathcal{B} + \mathcal{D}) = (\mathcal{A} \oplus \mathcal{B}) + (\mathcal{A} \oplus \mathcal{D})$$

$$\alpha(\mathcal{A} \otimes \infty) = (\alpha\mathcal{A}) \otimes \infty$$

shape rule: $\max(n_i, m_i)$ is output dimension

Broadcasting is fundamental in:

- neural network layers,
- attention mechanisms,
- normalization layers,
- GPU tensor fusion.

Implementation Principle

If a dimension is 1, repeat along that axis:

$$\mathcal{C}_{i,j,k} = \mathcal{A}_{i \bmod n_1, j \bmod n_2, k \bmod n_3} + \mathcal{B}_{i \bmod m_1, j \bmod m_2, k \bmod m_3}$$

Programming Task

Write a Python function that broadcasts two 3D tensors for element-wise addition.

```
def E07(A, B):
    # A shape: a1 x a2 x a3
    # B shape: b1 x b2 x b3

    a1, a2, a3 = len(A), len(A[0]), len(A[0][0])
    b1, b2, b3 = len(B), len(B[0]), len(B[0][0])

    # output shape = max along each dimension
    c1 = max(a1, b1)
    c2 = max(a2, b2)
    c3 = max(a3, b3)

    C = [[[0 for _ in range(c3)]
          for _ in range(c2)]
          for _ in range(c1)]

    for i in range(c1):
        for j in range(c2):
            for k in range(c3):
                # modulo indexing handles broadcasting
                ai = A[i % a1][j % a2][k % a3]
                bi = B[i % b1][j % b2][k % b3]
                C[i][j][k] = ai + bi

    return C
```

63 T03 – E08 : Tensor Indexing Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}.$$

Tensor indexing is the operation of selecting a subset of elements by specifying index values along one or more axes.

Basic indexing:

$$\mathcal{A}_{i,:,:} \in \mathbb{R}^{m \times p}$$

(select slice along the first dimension)

$$\mathcal{A}_{:,:,j} \in \mathbb{R}^{n \times p}$$

(select slice along the second dimension)

$$\mathcal{A}_{:,:,:,k} \in \mathbb{R}^{n \times m}$$

(select slice along the third dimension)

Single element:

$$\mathcal{A}_{i,j,k} \text{ is a scalar}$$

Row slice:

$$\mathcal{A}_{i, j_1:j_2, k_1:k_2}$$

Indexing allows:

- selecting sub-tensors,
- slicing ranges,
- extracting hyperplanes,
- reducing dimensions.

Indexing Rules

For any axis:

$$\text{index} \in \{0, 1, \dots, n_i - 1\}$$

Ranges:

$$a : b \rightarrow \{a, a + 1, \dots, b - 1\}$$

Full axis:

$$: \rightarrow \text{all indices}$$

Examples

Select the i -th matrix:

$$\mathcal{A}_{i,:,:}$$

Select the k -th feature map:

$$\mathcal{A}_{:,:,k}$$

Select a subvolume:

$$\mathcal{A}_{i_1:i_2, j_1:j_2, k_1:k_2}$$

Implementation Principle

To extract slice (i, j, k) :

index into tensor using nested loops or direct lookup

Programming Task

Write a Python function that extracts a sub-tensor given index ranges.

```
def E08(A, i1, i2, j1, j2, k1, k2):
    # A shape: n x m x p

    out = []

    for i in range(i1, i2):
        row_block = []
        for j in range(j1, j2):
            col_block = []
            for k in range(k1, k2):
                col_block.append(A[i][j][k])
            row_block.append(col_block)
        out.append(row_block)

    return out
```

64 T03 – E09 : Tensor Advanced Slicing Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}.$$

Advanced slicing is the operation of selecting arbitrary index sets from each dimension, not necessarily consecutive.

Instead of continuous ranges:

$$i_1 : i_2$$

we allow index lists:

$$I = (i_1, i_2, \dots, i_s)$$

$$J = (j_1, j_2, \dots, j_t)$$

$$K = (k_1, k_2, \dots, k_u)$$

The advanced slice is:

$$\mathcal{B}_{a,b,c} = \mathcal{A}_{I[a], J[b], K[c]}$$

Shape:

$$\mathcal{B} \in \mathbb{R}^{|I| \times |J| \times |K|}$$

This allows:

- selecting non-contiguous elements,
- gathering specific rows/columns/features,
- reordering axes,
- masking and indexing with arbitrary patterns.

Examples

Select odd slices:

$$I = (0, 2, 4), \quad J = (1, 3), \quad K = (0, 3, 5)$$

Gather specific locations:

$$\mathcal{A}_{(2,5),(1),(0,4,7)}$$

Permutation slice:

$$I = (3, 1, 0, 2)$$

Properties

$$\text{adv_slice}(\alpha \mathcal{A}, I, J, K) = \alpha \cdot \text{adv_slice}(\mathcal{A}, I, J, K)$$

$$\text{adv_slice}(\mathcal{A} + \mathcal{B}, I, J, K) = \text{adv_slice}(\mathcal{A}, I, J, K) + \text{adv_slice}(\mathcal{B}, I, J, K)$$

Implementation Principle

Use index lists instead of ranges:

$$\mathcal{B}_{a,b,c} = \mathcal{A}_{I[a], J[b], K[c]}$$

Programming Task

Write a Python function that extracts a sub-tensor using advanced slicing.

```
def E09(A, I, J, K):
    # I, J, K are lists of indices
    out = []

    for i in I:
        row_block = []
        for j in J:
            col_block = []
            for k in K:
                col_block.append(A[i][j][k])
            row_block.append(col_block)
        out.append(row_block)

    return out
```

65 T03 – E10 : Tensor Padding Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}.$$

The **padding operation** expands the tensor by adding extra elements around its borders.

Given padding amounts:

$$(p_1^-, p_1^+), \quad (p_2^-, p_2^+), \quad (p_3^-, p_3^+)$$

for each dimension, the padded tensor has shape:

$$(n + p_1^- + p_1^+) \times (m + p_2^- + p_2^+) \times (p + p_3^- + p_3^+)$$

Definition:

$$\mathcal{B}_{i,j,k} = \begin{cases} \mathcal{A}_{i-p_1^-, j-p_2^-, k-p_3^-}, & \text{if inside bounds} \\ 0, & \text{otherwise} \end{cases}$$

Padding types:

- Zero padding
- Constant padding
- Reflect padding
- Edge padding

Here we implement **zero padding**.

Properties

$$\begin{aligned} \text{pad}(\alpha \mathcal{A}) &= \alpha \text{pad}(\mathcal{A}) \\ \text{pad}(\mathcal{A} + \mathcal{B}) &= \text{pad}(\mathcal{A}) + \text{pad}(\mathcal{B}) \end{aligned}$$

Implementation Principle

Fill a larger tensor with zeros, then copy original values:

$$\mathcal{B}_{i+p_1^-, j+p_2^-, k+p_3^-} = \mathcal{A}_{i,j,k}$$

Programming Task

Write a Python function that applies zero padding to a 3D tensor.

```
def E10(A, pad1, pad2, pad3):  
    # pad1 = (left, right)  
    # pad2 = (top, bottom)  
    # pad3 = (front, back)  
  
    n = len(A)  
    m = len(A[0])  
    p = len(A[0][0])  
  
    p1l, p1r = pad1  
    p2l, p2r = pad2  
    p3l, p3r = pad3  
  
    N = n + p1l + p1r  
    M = m + p2l + p2r  
    P = p + p3l + p3r
```

```

B = [[[0 for _ in range(P)] for _ in range(M)] for _ in range(N)]

for i in range(n):
    for j in range(m):
        for k in range(p):
            B[i + p11][j + p21][k + p31] = A[i][j][k]

return B

```

66 T03 – E11 : Tensor Cropping Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} = \mathcal{A}_{ijk} \in \mathbb{R}^{n \times m \times p}.$$

Tensor cropping extracts a smaller sub-tensor by selecting a continuous region from each dimension. Given crop ranges:

$$i_1 : i_2, \quad j_1 : j_2, \quad k_1 : k_2$$

The cropped tensor is:

$$\mathcal{B}_{a,b,c} = \mathcal{A}_{i_1+a, j_1+b, k_1+c}$$

Resulting shape:

$$(i_2 - i_1) \times (j_2 - j_1) \times (k_2 - k_1)$$

This operation is fundamental in:

- computer vision (image cropping),
- 3D data processing,
- tensor region extraction,
- windowing for attention models.

Properties

$$\begin{aligned} \text{crop}(\alpha \mathcal{A}) &= \alpha \cdot \text{crop}(\mathcal{A}) \\ \text{crop}(\mathcal{A} + \mathcal{B}) &= \text{crop}(\mathcal{A}) + \text{crop}(\mathcal{B}) \end{aligned}$$

Cropping is the inverse of padding when values are preserved.

Implementation Principle

For all valid indices:

$$\mathcal{B}_{a,b,c} = \mathcal{A}_{i_1+a, j_1+b, k_1+c}$$

Programming Task

Write a Python function that extracts a cropped 3D tensor.

```

def E11(A, i1, i2, j1, j2, k1, k2):
    # A shape: n x m x p

    B = []

    for i in range(i1, i2):
        row_block = []
        for j in range(j1, j2):
            for k in range(k1, k2):
                row_block.append(A[i][j][k])
        B.append(row_block)

```

```

    col_block = []
    for k in range(k1, k2):
        col_block.append(A[i][j][k])
    row_block.append(col_block)
B.append(row_block)

return B

```

67 T03 – E12 : Tensor Stacking Law

Mathematical Definition

Let two 3D tensors with identical shapes:

$$\mathcal{A} \in \mathbb{R}^{n \times m \times p}, \quad \mathcal{B} \in \mathbb{R}^{n \times m \times p}.$$

The **stacking operation** creates a new tensor by adding a new dimension and placing the inputs along that dimension.

Stack along a new axis (0th axis):

$$\mathcal{C} = \text{stack}(\mathcal{A}, \mathcal{B}) \in \mathbb{R}^{2 \times n \times m \times p}$$

Definition:

$$\mathcal{C}_{0,i,j,k} = \mathcal{A}_{i,j,k} \quad \mathcal{C}_{1,i,j,k} = \mathcal{B}_{i,j,k}$$

General stacking of tensors

$$\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \dots, \mathcal{A}^{(r)}$$

produces a tensor of shape:

$$r \times n \times m \times p$$

Properties

$$\begin{aligned} \text{stack}(\alpha\mathcal{A}, \alpha\mathcal{B}) &= \alpha \cdot \text{stack}(\mathcal{A}, \mathcal{B}) \\ \text{stack}(\mathcal{A} + \mathcal{B}, \mathcal{C} + \mathcal{D}) &= \text{stack}(\mathcal{A}, \mathcal{C}) + \text{stack}(\mathcal{B}, \mathcal{D}) \\ \text{shape rule: } (n, m, p) &\rightarrow (r, n, m, p) \end{aligned}$$

Stacking is essential in:

- building mini-batches,
- grouping feature maps,
- forming 4D tensors for CNNs,
- constructing sequences for transformers.

Implementation Principle

For two tensors:

$$\mathcal{C}_{0,:,:,:} = \mathcal{A}, \quad \mathcal{C}_{1,:,:,:} = \mathcal{B}$$

Programming Task

Write a Python function that stacks two 3D tensors along a new dimension.

```

def E12(A, B):
    # A and B have shape: n x m x p
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])

```

```

# Output shape: 2 x n x m x p
C = [
    [[A[i][j][k] for k in range(p)] for j in range(m)]
        for i in range(n)
], [
    [[B[i][j][k] for k in range(p)] for j in range(m)]
        for i in range(n)
]

return C

```

68 T03 – E13 : Tensor Splitting Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{n \times m \times p}.$$

Tensor splitting divides a tensor into multiple smaller tensors along a chosen dimension.
Given a split along the first dimension:

$$n = s_1 + s_2 + \dots + s_r$$

We produce:

$$\mathcal{A}^{(1)} \in \mathbb{R}^{s_1 \times m \times p}, \quad \mathcal{A}^{(2)} \in \mathbb{R}^{s_2 \times m \times p}, \quad \dots, \quad \mathcal{A}^{(r)} \in \mathbb{R}^{s_r \times m \times p}$$

Definition:

$$\mathcal{A}_{i,j,k}^{(k)} = \mathcal{A}_{(\sum_{t < k} s_t) + i, j, k}$$

Splitting allows:

- minibatching,
- partitioning data,
- chunking large tensors,
- windowing for attention models.

Example

Split along axis 0:

$$\mathcal{A} \rightarrow (\mathcal{A}_{0:s_1}, \mathcal{A}_{s_1:s_1+s_2}, \dots)$$

Properties

$$\begin{aligned} \text{split}(\alpha \mathcal{A}) &= \alpha \cdot \text{split}(\mathcal{A}) \\ \text{concat}(\text{split}(\mathcal{A})) &= \mathcal{A} \\ \sum_k s_k &= n \end{aligned}$$

Implementation Principle

Use continuous ranges:

$$\mathcal{A}^{(k)} = \mathcal{A}_{s_{k-1}:s_k}$$

Programming Task

Write a Python function that splits a 3D tensor along axis 0 into chunks of given sizes.

```
def E13(A, sizes):
    # sizes = [s1, s2, s3, ...]
    n = len(A)
    out = []
    start = 0

    for s in sizes:
        end = start + s
        chunk = []

        for i in range(start, end):
            row_block = []
            for j in range(len(A[0])):
                col_block = []
                for k in range(len(A[0][0])):
                    col_block.append(A[i][j][k])
                row_block.append(col_block)
            chunk.append(row_block)

        out.append(chunk)
        start = end

    return out
```

69 T03 – E14 : Tensor Chunking Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{n \times m \times p}.$$

Tensor chunking divides a tensor into equal-sized blocks along a chosen dimension. Given a chunk size c , the number of chunks along axis 0 is:

$$r = \left\lfloor \frac{n}{c} \right\rfloor$$

Each chunk is:

$$\mathcal{A}^{(k)} \in \mathbb{R}^{c \times m \times p}$$

Definition:

$$\mathcal{A}_{i,j,k}^{(k)} = \mathcal{A}_{kc+i, j, k}$$

Chunking is used in:

- sequence processing,
- minibatch creation,
- splitting long feature maps,
- parallel computation on GPUs.

Example

If:

$$n = 12, \quad c = 3$$

then:

$$r = 4 \quad \Rightarrow \quad 4 \text{ chunks each of size } 3 \times m \times p$$

Properties

$$\begin{aligned}\text{concat}(\text{chunk}(\mathcal{A}, c)) &= \mathcal{A} \\ \text{chunk}(\alpha \mathcal{A}, c) &= \alpha \cdot \text{chunk}(\mathcal{A}, c)\end{aligned}$$

Implementation Principle

Fixed step along axis 0:

$$\mathcal{A}^{(k)} = \mathcal{A}_{kc:(k+1)c, :, :}$$

Programming Task

Write a Python function that chunks a tensor along axis 0.

```
def E14(A, c):
    # chunk size c
    n = len(A)
    m = len(A[0])
    p = len(A[0][0])

    chunks = []
    r = n // c  # number of full chunks

    for k in range(r):
        start = k * c
        end = start + c

        block = []
        for i in range(start, end):
            row_block = []
            for j in range(m):
                col_block = []
                for z in range(p):
                    col_block.append(A[i][j][z])
                row_block.append(col_block)
            block.append(row_block)

        chunks.append(block)

    return chunks
```

70 Tensor Shape Transformation Laws

The purpose of **T04** is to teach how tensor shapes can be transformed, manipulated, and reorganized without changing the underlying numerical values. Shape transformations are one of the most important concepts in tensor mathematics, deep learning, and GPU programming.

In modern AI systems, nearly every model — from CNNs to Transformers — depends heavily on reshaping operations such as flattening, squeezing, permuting, expanding, and repeating tensors.

Each exercise in **T04** presents a mathematical law that explains how a tensor's shape is transformed, followed by a Python implementation using pure lists (no PyTorch yet). The student reads the law, understands the transformation, and then writes the corresponding code.

This track includes the following laws:

- E00 – Tensor Reshape Law
- E01 – Tensor View Law
- E02 – Tensor Permute Law

- E03 – Tensor Transpose Law
- E04 – Tensor Expand Law
- E05 – Tensor Repeat Law
- E06 – Tensor Flip Law
- E07 – Tensor Rotate Law
- E08 – Tensor Squeeze Law
- E09 – Tensor Unsqueeze Law
- E10 – Tensor Flatten Law
- E11 – Tensor Unflatten Law

By completing T04, the student will learn how to control and manipulate tensor dimensions, build correct model inputs, and properly prepare data for advanced architectures such as CNNs, RNNs, and Transformers.

71 T04 – E00 : Tensor Reshape Law

Mathematical Definition

Let a tensor with total number of elements:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}, \quad N = d_1 d_2 \dots d_k.$$

The **reshape operation** changes the shape of the tensor without modifying its data, producing a new tensor with dimensions:

$$\mathcal{B} \in \mathbb{R}^{e_1 \times e_2 \times \dots \times e_m}$$

such that:

$$e_1 e_2 \dots e_m = N.$$

The mapping is defined by linear index:

Flatten original tensor:

$$\text{flat}[t] = \mathcal{A}_{i_1, i_2, \dots, i_k}$$

where:

$$t = i_1(d_2 d_3 \dots d_k) + i_2(d_3 \dots d_k) + \dots + i_k$$

Then reshape:

$$\mathcal{B}_{j_1, j_2, \dots, j_m} = \text{flat}[u]$$

where:

$$u = j_1(e_2 e_3 \dots e_m) + j_2(e_3 \dots e_m) + \dots + j_m$$

Thus reshape preserves order (row-major).

Properties

$$\begin{aligned} \text{reshape}(\text{reshape}(\mathcal{A})) &= \mathcal{A} \\ \text{reshape}(\alpha \mathcal{A}) &= \alpha \cdot \text{reshape}(\mathcal{A}) \\ \prod_i d_i &= \prod_j e_j \end{aligned}$$

Implementation Principle

Flatten → rebuild shape:

$$\mathcal{A} \rightarrow \text{flat} \rightarrow \mathcal{B}$$

Programming Task

Write a Python function that reshapes a 3D tensor into new dimensions (e_1, e_2, e_3) assuming the total number of elements matches.

```
def E00(A, e1, e2, e3):
    # Flatten the input tensor A
    flat = []
    for i in range(len(A)):
        for j in range(len(A[0])):
            for k in range(len(A[0][0])):
                flat.append(A[i][j][k])

    # Build new tensor
    out = [[[0 for _ in range(e3)] for _ in range(e2)] for _ in range(e1)]

    idx = 0
    for i in range(e1):
        for j in range(e2):
            for k in range(e3):
                out[i][j][k] = flat[idx]
                idx += 1

    return out
```

72 T04 – E01 : Tensor View Law

Mathematical Definition

Let a tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}, \quad N = d_1 d_2 \dots d_k.$$

A **view** is a new tensor that shares the same underlying data as \mathcal{A} , but with a different shape:

$$\mathcal{B} \in \mathbb{R}^{e_1 \times e_2 \times \dots \times e_m}$$

and satisfies:

$$e_1 e_2 \dots e_m = N.$$

A view does **not** copy data. It only remaps the index access function.

Given a linear index:

$$t = i_1(d_2 d_3 \dots d_k) + i_2(d_3 \dots d_k) + \dots + i_k,$$

the same element is accessed in the view as:

$$\mathcal{B}_{j_1, j_2, \dots, j_m} = \mathcal{A}_{i_1, i_2, \dots, i_k}$$

where

$$t = j_1(e_2 e_3 \dots e_m) + j_2(e_3 \dots e_m) + \dots + j_m.$$

Thus:

$$\text{view}(\mathcal{A}) = \text{reshape}(\mathcal{A}) \text{ without copying.}$$

Properties

$$\text{view}(\text{view}(\mathcal{A})) = \mathcal{A}$$

\mathcal{A} and $\text{view}(\mathcal{A})$ share memory

$$\prod_i d_i = \prod_j e_j$$

Conceptual Principle

view = reshape without reallocation

Programming Task

Since Python lists cannot share memory the way tensors do, we simulate a view by building an accessor function instead of copying data.

```
def E01(A, e1, e2, e3):
    # Create flat reference to the data
    flat = []
    for i in range(len(A)):
        for j in range(len(A[0])):
            for k in range(len(A[0][0])):
                flat.append(A[i][j][k])

    # Return a function that reads data in the new shape
    def view(i, j, k):
        # compute linear index
        idx = i * (e2 * e3) + j * e3 + k
        return flat[idx]

    return view
```

73 T04 – E02 : Tensor Permute Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

A **permutation** of dimensions rearranges the axes of the tensor according to a permutation:

$$\sigma = (\sigma_1, \sigma_2, \sigma_3) \in S_3$$

where S_3 is the permutation group of 3 elements.

The permuted tensor:

$$\mathcal{B} = \text{permute}(\mathcal{A}, \sigma)$$

has shape:

$$(d_{\sigma_1}, d_{\sigma_2}, d_{\sigma_3})$$

Definition:

$$\mathcal{B}_{i,j,k} = \mathcal{A} \begin{cases} i & \text{if } \sigma_1 = 1 \\ j & \text{if } \sigma_1 = 2, \\ k & \text{if } \sigma_1 = 3 \end{cases} \begin{cases} i & \text{if } \sigma_2 = 1 \\ j & \text{if } \sigma_2 = 2, \\ k & \text{if } \sigma_2 = 3 \end{cases} \begin{cases} i & \text{if } \sigma_3 = 1 \\ j & \text{if } \sigma_3 = 2 \\ k & \text{if } \sigma_3 = 3 \end{cases}$$

Example:

$$\sigma = (2, 1, 3)$$

swaps dimension 1 and 2:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{j,i,k}$$

Properties

$$\text{permute}(\text{permute}(\mathcal{A}, \sigma), \sigma^{-1}) = \mathcal{A}$$

$$\text{permute}(\alpha \mathcal{A}, \sigma) = \alpha \cdot \text{permute}(\mathcal{A}, \sigma)$$

$$\text{shape rule: } (d_1, d_2, d_3) \rightarrow (d_{\sigma_1}, d_{\sigma_2}, d_{\sigma_3})$$

Implementation Principle

Apply index remapping:

$$(i, j, k) \mapsto (u, v, w)$$

where:

$$(u, v, w) = (x_{\sigma_1}, x_{\sigma_2}, x_{\sigma_3})$$

with:

$$(x_1, x_2, x_3) = (i, j, k)$$

Programming Task

Write a Python function that permutes a 3D tensor according to a permutation tuple (s1, s2, s3).

```
def E02(A, s1, s2, s3):
    # s1, s2, s3 is a permutation of (0,1,2)

    dims = [len(A), len(A[0]), len(A[0][0])]
    out_dims = [dims[s1], dims[s2], dims[s3]]

    # create output tensor
    B = [[[0 for _ in range(out_dims[2])]
          for _ in range(out_dims[1])]
          for _ in range(out_dims[0])]

    for i in range(out_dims[0]):
        for j in range(out_dims[1]):
            for k in range(out_dims[2]):

                # original index
                src = [None, None, None]
                src[s1] = i
                src[s2] = j
                src[s3] = k

                B[i][j][k] = A[src[0]][src[1]][src[2]]
```

return B

74 T04 – E03 : Tensor Transpose Law

Mathematical Definition

For a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3},$$

a **transpose** swaps **two** dimensions while keeping the third dimension fixed.

Let the transpose swap axes a and b , where:

$$(a, b) \in \{(1, 2), (1, 3), (2, 3)\}.$$

The transposed tensor:

$$\mathcal{B} = \text{transpose}(\mathcal{A}, a, b)$$

has shape:

$$(d'_1, d'_2, d'_3)$$

where:

$$d'_a = d_b, \quad d'_b = d_a, \quad d'_c = d_c.$$

Definition:

$$\mathcal{B}_{i,j,k} = \begin{cases} \mathcal{A}_{j,i,k}, & \text{if } (a,b) = (1,2) \\ \mathcal{A}_{k,j,i}, & \text{if } (a,b) = (1,3) \\ \mathcal{A}_{i,k,j}, & \text{if } (a,b) = (2,3) \end{cases}$$

Example (swap axis 1 and 2):

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{j,i,k}$$

Transpose is a special case of permutation:

$$\text{transpose}(a, b) = \text{permute}(\sigma)$$

with σ being a swap.

Properties

$$\begin{aligned} \text{transpose}(\text{transpose}(\mathcal{A}, a, b), a, b) &= \mathcal{A} \\ \text{transpose}(\alpha \mathcal{A}, a, b) &= \alpha \cdot \text{transpose}(\mathcal{A}, a, b) \\ \text{transpose}(\mathcal{A}^T) &= \mathcal{A} \end{aligned}$$

Implementation Principle

Swap two axes:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{u,v,w}$$

where:

$$(u, v, w) = \text{swap}(i, j, k)$$

Programming Task

Write a Python function that transposes a 3D tensor by swapping two axes.

```
def E03(A, a, b):
    # a and b are axis indices: 0,1,2

    dims = [len(A), len(A[0]), len(A[0][0])]

    # output dims after swapping
    out_dims = dims.copy()
    out_dims[a], out_dims[b] = out_dims[b], out_dims[a]

    # create output tensor
    B = [[[0 for _ in range(out_dims[2])] for _ in range(out_dims[1])] for _ in range(out_dims[0])]

    for i in range(out_dims[0]):
        for j in range(out_dims[1]):
            for k in range(out_dims[2]):

                # original index before swap
                src = [i, j, k]
                src[a], src[b] = src[b], src[a]

                B[i][j][k] = A[src[0]][src[1]][src[2]]
```

return B

75 T04 – E04 : Tensor Expand Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

The **expand operation** increases the size of a tensor by repeating elements along specific axes, without copying the full underlying data (conceptually).

Given target dimensions:

$$(e_1, e_2, e_3)$$

such that:

$$e_i = d_i \quad \text{or} \quad d_i = 1.$$

Expanded tensor:

$$\mathcal{B} \in \mathbb{R}^{e_1 \times e_2 \times e_3}$$

Definition:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod d_1, j \bmod d_2, k \bmod d_3}$$

This is equivalent to broadcasting a tensor to a larger shape.

Example:

$$(1, 4, 1) \rightarrow (3, 4, 5)$$

Properties

$$\text{expand}(\mathcal{A}) = \text{broadcast}(\mathcal{A})$$

$$\text{expand}(\alpha \mathcal{A}) = \alpha \cdot \text{expand}(\mathcal{A})$$

$$e_i \geq d_i$$

Implementation Principle

Repeat values along axes with size 1:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod d_1, j \bmod d_2, k \bmod d_3}$$

Programming Task

Write a Python function that expands a 3D tensor from shape (d_1, d_2, d_3) to (e_1, e_2, e_3) .

```
def E04(A, e1, e2, e3):
    d1 = len(A)
    d2 = len(A[0])
    d3 = len(A[0][0])

    B = [[[0 for _ in range(e3)]
          for _ in range(e2)]
          for _ in range(e1)]

    for i in range(e1):
        for j in range(e2):
            for k in range(e3):
                B[i][j][k] = A[i % d1][j % d2][k % d3]

    return B
```

76 T04 – E05 : Tensor Repeat Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

The **repeat operation** duplicates the tensor a specified number of times along each axis to produce a larger tensor.

Given repeat factors:

$$(r_1, r_2, r_3) \in \mathbb{N}^3,$$

the output shape is:

$$(e_1, e_2, e_3) = (r_1 d_1, r_2 d_2, r_3 d_3).$$

Definition:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod d_1, j \bmod d_2, k \bmod d_3}.$$

Repeat is different from expand:

- **expand** grows only when a dimension is 1.
- **repeat** grows by multiplying dimensions.

Example:

$$(2, 3, 1) \text{ with repeats } (2, 1, 4)$$

becomes:

$$(4, 3, 4)$$

Properties

$$\begin{aligned}\text{repeat}(\alpha \mathcal{A}) &= \alpha \cdot \text{repeat}(\mathcal{A}) \\ \text{repeat}(\mathcal{A}, 1, 1, 1) &= \mathcal{A} \\ \text{repeat}(\text{repeat}(\mathcal{A}, r), s) &= \text{repeat}(\mathcal{A}, r \cdot s)\end{aligned}$$

Implementation Principle

Fill a larger tensor by periodic indexing:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i \bmod d_1, j \bmod d_2, k \bmod d_3}$$

Programming Task

Write a Python function that repeats a 3D tensor along the three axes.

```
def E05(A, r1, r2, r3):  
    d1 = len(A)  
    d2 = len(A[0])  
    d3 = len(A[0][0])  
  
    e1 = r1 * d1  
    e2 = r2 * d2  
    e3 = r3 * d3  
  
    B = [[[0 for _ in range(e3)]  
          for _ in range(e2)]  
          for _ in range(e1)]  
  
    for i in range(e1):  
        for j in range(e2):  
            for k in range(e3):  
                B[i][j][k] = A[i % d1][j % d2][k % d3]  
  
    return B
```

77 T04 – E06 : Tensor Flip Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

The **flip operation** reverses the order of elements along one or more axes.

Given a flip along axis $a \in \{1, 2, 3\}$, the flipped tensor:

$$\mathcal{B} = \text{flip}(\mathcal{A}, a)$$

is defined by:

$$\mathcal{B}_{i,j,k} = \mathcal{A}_{i',j',k'}$$

where:

$$(i', j', k') = \begin{cases} (d_1 - 1 - i, j, k) & \text{if axis} = 1 \\ (i, d_2 - 1 - j, k) & \text{if axis} = 2 \\ (i, j, d_3 - 1 - k) & \text{if axis} = 3 \end{cases}$$

Example:

$$\text{flip}(\mathcal{A}, 1) \Rightarrow \text{reverse rows}$$

Multi-Axis Flip

For multiple axes:

$$\text{flip}(\mathcal{A}, \{a_1, a_2, \dots\})$$

the mapping applies each flip rule.

Properties

$$\begin{aligned} \text{flip}(\text{flip}(\mathcal{A}, a), a) &= \mathcal{A} \\ \text{flip}(\alpha \mathcal{A}, a) &= \alpha \cdot \text{flip}(\mathcal{A}, a) \\ \text{flip} &\text{ is its own inverse} \end{aligned}$$

Implementation Principle

Reverse the index along selected axis:

$$i' = d_a - 1 - i$$

Programming Task

Write a Python function that flips a 3D tensor along any axis.

```
def E06(A, axis):
    d1 = len(A)
    d2 = len(A[0])
    d3 = len(A[0][0])

    B = [[[0 for _ in range(d3)]
          for _ in range(d2)]
          for _ in range(d1)]

    for i in range(d1):
        for j in range(d2):
            for k in range(d3):

                if axis == 0:
                    B[i][j][k] = A[d1 - 1 - i][j][k]
```

```

    elif axis == 1:
        B[i][j][k] = A[i][d2 - 1 - j][k]
    elif axis == 2:
        B[i][j][k] = A[i][j][d3 - 1 - k]

return B

```

78 T04 – E07 : Tensor Rotate Law

Mathematical Definition

Let a 3D tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

The **rotation operation** rotates the tensor by 90° , 180° , or 270° around a chosen axis. A rotation is defined as a composition of:

- transpose (swapping dimensions)
- flip (reversing an axis)

For a rotation around axis 0:

$$\text{rotate}_0(90^\circ) : \quad \mathcal{B}_{i,j,k} = \mathcal{A}_{i, d_3-1-k, j}$$

For a rotation around axis 1:

$$\text{rotate}_1(90^\circ) : \quad \mathcal{B}_{i,j,k} = \mathcal{A}_{d_1-1-k, j, i}$$

For a rotation around axis 2:

$$\text{rotate}_2(90^\circ) : \quad \mathcal{B}_{i,j,k} = \mathcal{A}_{j, d_2-1-i, k}$$

Higher rotations:

$$\begin{aligned} \text{rotate}(180^\circ) &= \text{rotate}(90^\circ) \circ \text{rotate}(90^\circ) \\ \text{rotate}(270^\circ) &= \text{rotate}(90^\circ) \circ \text{rotate}(180^\circ) \end{aligned}$$

Properties

$$\begin{aligned} \text{rotate}(360^\circ) &= \mathcal{A} \\ \text{rotate}(90^\circ) \circ \text{rotate}(270^\circ) &= \mathcal{A} \\ \text{rotate}(\alpha \mathcal{A}) &= \alpha \cdot \text{rotate}(\mathcal{A}) \end{aligned}$$

Implementation Principle

Rotation is implemented via:

$$\text{rotate} = \text{transpose} + \text{flip}$$

Programming Task

Write a Python function that rotates a 3D tensor by 90° , 180° , or 270° around a specific axis.

```

def E07(A, axis, angle):
    # axis = 0,1,2
    # angle = 90, 180, 270

    # First compute 90-degree rotation
    def rot90(A, axis):
        d1 = len(A)
        d2 = len(A[0])

```

```

d3 = len(A[0][0])

B = [[[0 for _ in range(d3)]
      for _ in range(d2)]
      for _ in range(d1)]

for i in range(d1):
    for j in range(d2):
        for k in range(d3):
            if axis == 0:
                B[i][j][k] = A[i][d3 - 1 - k][j]
            elif axis == 1:
                B[i][j][k] = A[d1 - 1 - k][j][i]
            elif axis == 2:
                B[i][j][k] = A[j][d2 - 1 - i][k]
return B

# Apply rotation appropriate number of times
if angle == 90:
    return rot90(A, axis)
elif angle == 180:
    return rot90(rot90(A, axis), axis)
elif angle == 270:
    return rot90(rot90(rot90(A, axis), axis), axis)

```

79 T04 – E08 : Tensor Squeeze Law

Mathematical Definition

Let a tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}.$$

A **squeeze operation** removes all dimensions whose size is equal to 1.

Formally:

$$\text{squeeze}(\mathcal{A}) \in \mathbb{R}^{d_{i_1} \times d_{i_2} \times \dots}$$

where:

$$d_{i_j} \neq 1.$$

If a dimension equals 1 (e.g. $d_a = 1$), it is removed:

$$\mathcal{A} \in \mathbb{R}^{1 \times m \times 1 \times p} \Rightarrow \text{squeeze}(\mathcal{A}) \in \mathbb{R}^{m \times p}$$

If the tensor has no dimension equal to 1, squeeze does nothing.

Properties

$$\begin{aligned} \text{squeeze}(\text{squeeze}(\mathcal{A})) &= \text{squeeze}(\mathcal{A}) \\ \text{squeeze}(\alpha \mathcal{A}) &= \alpha \cdot \text{squeeze}(\mathcal{A}) \end{aligned}$$

Conceptual Principle

`squeeze` removes unnecessary singleton dimensions.

Implementation Principle

New tensor shape:

$$(d_1, d_2, \dots, d_k) \rightarrow (d_{i_1}, d_{i_2}, \dots) \quad \text{where } d_{i_j} \neq 1$$

Programming Task

Write a Python function that squeezes a 3D tensor by removing any dimension of size 1.

```
def E08(A):
    d1 = len(A)
    d2 = len(A[0])
    d3 = len(A[0][0])

    # case 1: all dims > 1
    if d1 > 1 and d2 > 1 and d3 > 1:
        return A

    # case 2: remove first dim
    if d1 == 1:
        A = A[0]
        d2 = len(A)
        d3 = len(A[0])

    # case 3: remove second dim
    if d2 == 1:
        A = [row[0] for row in A]
        d3 = len(A[0]) if d1 > 1 else len(A)

    # case 4: remove third dim
    if d3 == 1:
        if isinstance(A[0][0], list):
            A = [[row for row in col] for col in A]
        else:
            A = [[val for val in row] for row in A]

    return A
```

80 T04 – E09 : Tensor Unsqueeze Law

Mathematical Definition

Let a tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}.$$

The **unsqueeze operation** inserts a new dimension of size 1 at a specified axis.
Given axis a , the resulting tensor:

$$\mathcal{B} = \text{unsqueeze}(\mathcal{A}, a)$$

has shape:

$$(d_1, \dots, d_a, 1, d_{a+1}, \dots, d_k).$$

For example:

$$\begin{aligned} \mathcal{A} \in \mathbb{R}^{m \times p} \quad \Rightarrow \quad \text{unsqueeze}(\mathcal{A}, 0) &\in \mathbb{R}^{1 \times m \times p} \\ \text{unsqueeze}(\mathcal{A}, 2) &\in \mathbb{R}^{m \times p \times 1} \end{aligned}$$

Properties

$$\begin{aligned} \text{unsqueeze}(\alpha \mathcal{A}) &= \alpha \cdot \text{unsqueeze}(\mathcal{A}) \\ \text{squeeze}(\text{unsqueeze}(\mathcal{A}, a)) &= \mathcal{A} \end{aligned}$$

Implementation Principle

Insert a new dimension:

$$\mathcal{B}_{i_1, \dots, i_a, 0, i_{a+1}, \dots} = \mathcal{A}_{i_1, \dots, i_a, \dots}$$

Programming Task

Write a Python function that unsqueezes a 3D tensor at axis 0, 1, or 2.

```
def E09(A, axis):
    # A is assumed to be 3D: d1 x d2 x d3
    d1 = len(A)
    d2 = len(A[0])
    d3 = len(A[0][0])

    # axis 0: new shape (1,d1,d2,d3)
    if axis == 0:
        return [A]

    # axis 1: new shape (d1,1,d2,d3)
    if axis == 1:
        return [[row for row in A[i]] for i in range(d1)]

    # axis 2: new shape (d1,d2,1,d3) | simulate by adding level
    if axis == 2:
        B = []
        for i in range(d1):
            block = []
            for j in range(d2):
                block.append([A[i][j]]) # new dimension inserted
            B.append(block)
        return B

    # axis 3: new shape (d1,d2,d3,1)
    if axis == 3:
        B = []
        for i in range(d1):
            block = []
            for j in range(d2):
                row = []
                for k in range(d3):
                    row.append([A[i][j][k]])
                block.append(row)
            B.append(block)
        return B
```

81 T04 – E10 : Tensor Flatten Law

Mathematical Definition

Let a tensor:

$$\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}.$$

The **flatten operation** converts the tensor into a 1D vector by collapsing all axes into one axis.
Output shape:

$$\text{flatten}(\mathcal{A}) \in \mathbb{R}^N$$

where:

$$N = d_1 d_2 \dots d_k.$$

Definition using row-major order:

$$\text{flat}[t] = \mathcal{A}_{i_1, i_2, \dots, i_k}$$

with linear index:

$$t = i_1(d_2 d_3 \dots d_k) + i_2(d_3 \dots d_k) + \dots + i_k.$$

Properties

$$\begin{aligned} \text{flatten}(\text{reshape}(\mathcal{A})) &= \text{flatten}(\mathcal{A}) \\ \text{flatten}(\alpha \mathcal{A}) &= \alpha \cdot \text{flatten}(\mathcal{A}) \\ |\text{flat}| &= d_1 d_2 \dots d_k \end{aligned}$$

Conceptual Principle

`flatten` = remove all dimensions and list elements linearly.

Implementation Principle

Iterate through every index in row-major order and append to 1D vector.

Programming Task

Write a Python function that flattens a 3D tensor into a 1D list.

```
def E10(A):
    flat = []
    d1 = len(A)
    d2 = len(A[0])
    d3 = len(A[0][0])

    for i in range(d1):
        for j in range(d2):
            for k in range(d3):
                flat.append(A[i][j][k])

    return flat
```

82 T04 – E11 : Tensor Unflatten Law

Mathematical Definition

Let a 1D vector:

$$\text{flat} \in \mathbb{R}^N.$$

The **unflatten operation** converts this vector back into a tensor of shape:

$$\mathcal{B} \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_k}$$

such that:

$$N = d_1 d_2 \dots d_k.$$

Definition:

$$\mathcal{B}_{i_1, i_2, \dots, i_k} = \text{flat}[t]$$

where the linear index is:

$$t = i_1(d_2 d_3 \dots d_k) + i_2(d_3 \dots d_k) + \dots + i_k.$$

Unflatten is the exact inverse of flatten:

$$\text{unflatten}(\text{flatten}(\mathcal{A})) = \mathcal{A}.$$

Properties

$$\begin{aligned}\text{flatten}(\text{unflatten}(\text{flat})) &= \text{flat} \\ \text{unflatten}(\alpha \text{flat}) &= \alpha \cdot \text{unflatten}(\text{flat})\end{aligned}$$

Conceptual Principle

`unflatten` = reshape a linear vector into a multidimensional tensor.

Implementation Principle

Use row-major indexing to reconstruct the multi-dimensional tensor.

Programming Task

Write a Python function that unflattens a 1D list into a 3D tensor with shape (d_1, d_2, d_3) .

```
def E11(flat, d1, d2, d3):
    out = [[[0 for _ in range(d3)]
            for _ in range(d2)]
            for _ in range(d1)]

    idx = 0
    for i in range(d1):
        for j in range(d2):
            for k in range(d3):
                out[i][j][k] = flat[idx]
                idx += 1

    return out
```

83 Differential Calculus Laws (for PyTorch Autograd)

The purpose of **T05** is to introduce the mathematical foundations of **differential calculus for tensors**, and to show how these laws are implemented conceptually in modern deep learning frameworks such as PyTorch, TensorFlow, and JAX.

Deep learning depends entirely on one core mechanism:

Automatic Differentiation (Autograd)

which computes gradients through tensor operations. Every model, optimizer, and loss function in AI is powered by these differential laws.

In **T05**, each exercise focuses on a single mathematical derivative law. The student first learns the real mathematical equation, then rewrites it as pure Python code (without PyTorch), simulating the behavior of Autograd.

This track includes the following laws:

- E00 – Gradient Law
- E01 – Jacobian Matrix Law
- E02 – Hessian Matrix Law
- E03 – Divergence Law
- E04 – Curl Law
- E05 – Laplacian Law
- E06 – Partial Derivative Law

- E07 – Chain Rule Law
- E08 – Product Rule Law
- E09 – Quotient Rule Law
- E10 – Power Rule Law
- E11 – Exponential Derivative Law
- E12 – Logarithmic Derivative Law
- E13 – Sigmoid Derivative Law
- E14 – Tanh Derivative Law
- E15 – ReLU Derivative Law
- E16 – LeakyReLU Derivative Law
- E17 – Softmax Derivative Law
- E18 – Loss Gradient Law

By completing T05, the student will acquire the ability to compute gradients manually, understand how backpropagation works, and simulate essential parts of neural network training.

84 T05 – E00 : Gradient Law

Mathematical Definition

Let a scalar function:

$$f(x_1, x_2, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}.$$

The **gradient** of f is the vector of all partial derivatives:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n.$$

The gradient points in the direction of the greatest increase of the function. In machine learning, we use the negative gradient for optimization:

$$x_{\text{new}} = x - \eta \nabla f(x)$$

where η is the learning rate.

Example Function

Consider a function of two variables:

$$f(x, y) = x^2 + y^2.$$

Its gradient is:

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

Properties

$$\begin{aligned} \nabla(f + g) &= \nabla f + \nabla g \\ \nabla(\alpha f) &= \alpha \nabla f \\ \nabla(x^T A x) &= (A + A^T)x \\ \nabla f = 0 &\Rightarrow \text{critical point} \end{aligned}$$

Implementation Principle

To compute the gradient numerically, we approximate the derivative using finite differences:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}$$

where h is a very small value (e.g. 10^{-6}).

Programming Task

Write a Python function that computes the gradient of any function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ using numerical differentiation.

```
def E00(f, x, h=1e-6):
    """
    f : function that maps R^n -> R
    x : list of input values [x1, x2, ..., xn]
    h : small step for numerical derivative
    """

    n = len(x)
    grad = [0.0] * n

    for i in range(n):
        # create x+h and x form
        x_ph = x.copy()
        x_ph[i] += h

        # finite difference approximation
        grad[i] = (f(x_ph) - f(x)) / h

    return grad
```

85 T05 – E01 : Jacobian Matrix Law

Mathematical Definition

Let a vector-valued function:

$$\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad \mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

The **Jacobian matrix** is the matrix of all first-order partial derivatives:

$$J_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

Each row is the gradient of one output component:

$$J_{\mathbf{f}}(\mathbf{x})_{i,:} = \nabla f_i(\mathbf{x})^T$$

Example

Let:

$$\mathbf{f}(x, y) = \begin{bmatrix} x^2 + y \\ xy \end{bmatrix}$$

Then:

$$J_{\mathbf{f}} = \begin{bmatrix} 2x & 1 \\ y & x \end{bmatrix}$$

Properties

$$\begin{aligned} J_{f+g} &= J_f + J_g \\ J_{\alpha f} &= \alpha J_f \\ J_{g(f(x))} &= J_g(f(x)) \cdot J_f(x) \quad (\text{Chain Rule}) \\ J_f^T \mathbf{v} &= \text{vector-Jacobian product (VJP)} \end{aligned}$$

This is the basis of backpropagation.

Implementation Principle

Compute each partial derivative using finite differences:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_1, \dots, x_j + h, \dots, x_n) - f_i(x)}{h}$$

Programming Task

Write a Python function that computes the Jacobian of a vector-valued function using numerical differentiation.

```
def E01(f, x, h=1e-6):
    """
    f : function that maps R^n -> R^m (returns list of length m)
    x : list [x1, x2, ..., xn]
    h : small step for numerical derivative
    """

    n = len(x)
    y = f(x)                      # output of size m
    m = len(y)

    J = [[0.0 for _ in range(n)] for _ in range(m)]

    for j in range(n):
        x_ph = x.copy()
        x_ph[j] += h

        y_ph = f(x_ph)

        for i in range(m):
            J[i][j] = (y_ph[i] - y[i]) / h

    return J
```

86 T05 – E02 : Hessian Matrix Law

Mathematical Definition

Let a scalar function:

$$f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

The **Hessian matrix** of f is the matrix of all second-order partial derivatives:

$$H_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Each entry:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

The Hessian is symmetric if f is twice continuously differentiable:

$$H = H^T$$

Example

For:

$$f(x, y) = x^2 + xy + y^2$$

The gradient is:

$$\nabla f = \begin{bmatrix} 2x + y \\ x + 2y \end{bmatrix}$$

The Hessian is:

$$H_f = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

Properties

H_f is symmetric

$$H_{f+g} = H_f + H_g$$

$$H_{\alpha f} = \alpha H_f$$

$$H_f = \nabla(\nabla f)^T$$

A positive definite Hessian means a local minimum. A negative definite Hessian means a local maximum.

Implementation Principle

Using finite differences:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \approx \frac{f(x + he_i + he_j) - f(x + he_i) - f(x + he_j) + f(x)}{h^2}$$

where e_i is the unit vector with 1 in position i .

Programming Task

Write a Python function that computes the Hessian matrix numerically using second-order finite differences.

```
def E02(f, x, h=1e-5):
    """
    f : scalar function R^n -> R
    x : list [x1, x2, ..., xn]
    """

    n = len(x)
    H = [[0.0 for _ in range(n)] for _ in range(n)]

    fx = f(x)

    for i in range(n):
        for j in range(n):
            x_ij = x.copy()
            x_i_ = x.copy()
            x_j_ = x.copy()

            x_ij[i] += h
            x_ij[j] += h

            x_i_[i] += h
            x_j_[j] += h

            fij = f(x_ij)
            fi = f(x_i_)
            fj = f(x_j_)

            H[i][j] = (fij - fi - fj + fx) / (h * h)

    return H
```

87 T05 – E03 : Divergence Law

Mathematical Definition

Let a vector field:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} F_1(x_1, \dots, x_n) \\ F_2(x_1, \dots, x_n) \\ \vdots \\ F_n(x_1, \dots, x_n) \end{bmatrix} : \mathbb{R}^n \rightarrow \mathbb{R}^n.$$

The **divergence** of \mathbf{F} is a scalar field defined as the sum of the partial derivatives of each component with respect to its corresponding variable:

$$\operatorname{div} \mathbf{F} = \sum_{i=1}^n \frac{\partial F_i}{\partial x_i}.$$

Expanded in coordinates:

$$\operatorname{div} \mathbf{F} = \frac{\partial F_1}{\partial x_1} + \frac{\partial F_2}{\partial x_2} + \dots + \frac{\partial F_n}{\partial x_n}.$$

Example

For a 2D field:

$$\mathbf{F}(x, y) = \begin{bmatrix} xy \\ x^2 \end{bmatrix}$$

The divergence is:

$$\operatorname{div} \mathbf{F} = \frac{\partial(xy)}{\partial x} + \frac{\partial(x^2)}{\partial y} = y + 0 = y.$$

Properties

$$\begin{aligned}\operatorname{div}(\mathbf{F} + \mathbf{G}) &= \operatorname{div} \mathbf{F} + \operatorname{div} \mathbf{G} \\ \operatorname{div}(\alpha \mathbf{F}) &= \alpha \cdot \operatorname{div} \mathbf{F} \\ \operatorname{div}(\nabla f) &= \Delta f \quad (\text{Laplacian of } f)\end{aligned}$$

Implementation Principle

Using finite difference approximation:

$$\frac{\partial F_i}{\partial x_i} \approx \frac{F_i(x_1, \dots, x_i + h, \dots, x_n) - F_i(x_1, \dots, x_i, \dots, x_n)}{h}$$

Then sum over all components.

Programming Task

Write a Python function that computes the divergence of a vector field using numerical partial derivatives.

```
def E03(F, x, h=1e-6):
    """
    F : function R^n -> R^n (vector field)
    x : list [x1, x2, ..., xn]
    """

    n = len(x)
    Fx = F(x)  # returns vector of length n
    div = 0.0

    for i in range(n):
        # x + h e_i
        x_ph = x.copy()
        x_ph[i] += h

        # compute partial derivative of F_i w.r.t. x_i
        Fi_ph = F(x_ph)[i]
        Fi     = Fx[i]

        div += (Fi_ph - Fi) / h

    return div
```

88 T05 – E04 : Curl Law

Mathematical Definition

The curl applies only to 3-dimensional vector fields.

Let a vector field:

$$\mathbf{F}(x, y, z) = \begin{bmatrix} F_x(x, y, z) \\ F_y(x, y, z) \\ F_z(x, y, z) \end{bmatrix} : \mathbb{R}^3 \rightarrow \mathbb{R}^3.$$

The **curl** of \mathbf{F} is a new vector field defined as:

$$\nabla \times \mathbf{F} = \begin{bmatrix} \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \\ \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \\ \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \end{bmatrix}.$$

In determinant notation:

$$\nabla \times \mathbf{F} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_x & F_y & F_z \end{vmatrix}$$

Example

Let:

$$\mathbf{F}(x, y, z) = \begin{bmatrix} yz \\ xz \\ xy \end{bmatrix}$$

Then:

$$\nabla \times \mathbf{F} = \begin{bmatrix} x - x \\ y - y \\ z - z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Properties

$$\begin{aligned} \nabla \cdot (\nabla \times \mathbf{F}) &= 0 \\ \nabla \times (\nabla f) &= 0 \\ \nabla \times (\mathbf{F} + \mathbf{G}) &= \nabla \times \mathbf{F} + \nabla \times \mathbf{G} \\ \nabla \times (\alpha \mathbf{F}) &= \alpha(\nabla \times \mathbf{F}) \end{aligned}$$

Implementation Principle

Use central finite differences:

$$\frac{\partial F}{\partial x} \approx \frac{F(x+h) - F(x-h)}{2h}$$

Apply this to the 6 directional derivatives needed for curl.

Programming Task

Write a Python function that computes the curl of a vector field $\mathbf{F} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$.

```
def E04(F, x, h=1e-6):
    """
    F : function R^3 -> R^3
    x : [x, y, z]
    """
    x0, y0, z0 = x

    # compute central differences

    def dFdx(i):
        xp = F([x0 + h, y0, z0])[i]
        xm = F([x0 - h, y0, z0])[i]
        return (xp - xm) / (2*h)

    def dFdy(i):
        ...
```

```

yp = F([x0, y0 + h, z0])[i]
ym = F([x0, y0 - h, z0])[i]
return (yp - ym) / (2*h)

def dFdz(i):
    zp = F([x0, y0, z0 + h])[i]
    zm = F([x0, y0, z0 - h])[i]
    return (zp - zm) / (2*h)

curl_x = dFdz(2) - dFdz(1)
curl_y = dFdz(0) - dFdz(2)
curl_z = dFdz(1) - dFdz(0)

return [curl_x, curl_y, curl_z]

```

89 T05 – E05 : Laplacian Law

Mathematical Definition

The **Laplacian** of a scalar function

$$f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$$

is defined as the divergence of the gradient:

$$\Delta f = \nabla \cdot (\nabla f)$$

Expanded in coordinates:

$$\Delta f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \cdots + \frac{\partial^2 f}{\partial x_n^2}.$$

Example

For:

$$f(x, y) = x^2 + y^2$$

The second derivatives are:

$$\frac{\partial^2 f}{\partial x^2} = 2, \quad \frac{\partial^2 f}{\partial y^2} = 2.$$

Thus:

$$\Delta f = 2 + 2 = 4.$$

Properties

$$\begin{aligned} \Delta(f + g) &= \Delta f + \Delta g \\ \Delta(\alpha f) &= \alpha \Delta f \\ \Delta f &= \text{trace}(H_f) \\ \Delta f &= \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2} \end{aligned}$$

Relationship to Other Operators

$$\nabla f \quad (\text{gradient})$$

$$\nabla \cdot \mathbf{F} \quad (\text{divergence})$$

$$\Delta f = \nabla \cdot (\nabla f) \quad (\text{Laplacian})$$

The Laplacian measures how much a function “curves outward” from a point. This is used in diffusion models and heat propagation.

Implementation Principle

Use second-order finite differences:

$$\frac{\partial^2 f}{\partial x_i^2} \approx \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2}$$

and sum over all dimensions.

Programming Task

Write a Python function that computes the Laplacian numerically.

```
def E05(f, x, h=1e-4):
    """
    f : scalar function R^n -> R
    x : list [x1, x2, ..., xn]
    """

    n = len(x)
    fx = f(x)
    lap = 0.0

    for i in range(n):
        x_ph = x.copy()
        x_mh = x.copy()

        x_ph[i] += h
        x_mh[i] -= h

        lap += (f(x_ph) - 2*fx + f(x_mh)) / (h*h)

    return lap
```

90 T05 – E06 : Partial Derivative Law

Mathematical Definition

Let a multivariable function:

$$f(x_1, x_2, \dots, x_n) : \mathbb{R}^n \rightarrow \mathbb{R}.$$

The **partial derivative** of f with respect to x_i is defined as:

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}.$$

This measures how f changes when only one variable moves while the rest stay fixed.

Example

Let:

$$f(x, y) = x^2y + 3y.$$

Then:

$$\frac{\partial f}{\partial x} = 2xy, \quad \frac{\partial f}{\partial y} = x^2 + 3.$$

Properties

$$\frac{\partial}{\partial x_i}(f + g) = \frac{\partial f}{\partial x_i} + \frac{\partial g}{\partial x_i}$$

$$\frac{\partial}{\partial x_i}(\alpha f) = \alpha \frac{\partial f}{\partial x_i}$$

$$\frac{\partial}{\partial x_i}(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial}{\partial x_i}(x_i^k) = kx_i^{k-1}$$

Relationship to Other Operators

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \quad (\text{gradient})$$

$$H_f = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right] \quad (\text{Hessian})$$

$$\Delta f = \sum_i \frac{\partial^2 f}{\partial x_i^2} \quad (\text{Laplacian})$$

So partial derivatives are the building blocks of all tensor calculus laws.

Implementation Principle

Finite difference approximation:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x)}{h}$$

Programming Task

Write a Python function that computes the partial derivative of f with respect to x_i .

```
def E06(f, x, i, h=1e-6):
    """
    f : scalar function R^n -> R
    x : list [x1, x2, ..., xn]
    i : index of variable to differentiate (0-based)
    """

    x_ph = x.copy()
    x_ph[i] += h

    return (f(x_ph) - f(x)) / h
```

91 T05 – E07 : Chain Rule Law

Mathematical Definition

Let:

$$y = f(u), \quad u = g(x).$$

The **Chain Rule** states:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$

In the multivariable case:

$$\frac{\partial f}{\partial x_i} = \sum_j \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}.$$

Or in vector notation:

$$\nabla_x f = J_g(x)^T \nabla_u f.$$

Example

Let:

$$u = g(x) = x^2, \quad y = f(u) = 3u + 1.$$

Then:

$$\frac{dy}{du} = 3, \quad \frac{du}{dx} = 2x.$$

Thus:

$$\frac{dy}{dx} = 3 \cdot 2x = 6x.$$

Properties

Chain Rule is the basis of backpropagation

$$\begin{aligned} \nabla_x f &= J_g^T \nabla_u f \\ \frac{d}{dx} f(g(h(x))) &= f'(g(h(x))) g'(h(x)) h'(x) \end{aligned}$$

Relationship to Neural Networks

For a neural layer:

$$u = Wx + b, \quad y = \sigma(u)$$

The gradient flows as:

$$\nabla_x y = W^T \cdot \sigma'(u)$$

This is how gradients propagate backward in deep learning.

Implementation Principle

Numerical chain rule for:

$$f(g(x)).$$

$$\frac{d}{dx} f(g(x)) \approx \frac{f(g(x+h)) - f(g(x))}{h}.$$

Programming Task

Implement the chain rule numerically for scalar functions.

```
def E07(f, g, x, h=1e-6):
    """
    Computes derivative of f(g(x)) using numerical chain rule.

    f : function R -> R
    g : function R -> R
    """

    # compute g(x)
    gx = g(x)

    # derivative of f at g(x)
    dfdg = (f(gx + h) - f(gx)) / h

    # derivative of g at x
    dgdx = (g(x + h) - g(x)) / h

    # chain rule
    return dfdg * dgdx
```

92 T05 – E08 : Product Rule Law

Mathematical Definition

Let two differentiable functions:

$$f(x), \quad g(x).$$

The **Product Rule** states:

$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x).$$

In multivariable form:

$$\frac{\partial}{\partial x_i}(f \cdot g) = \frac{\partial f}{\partial x_i}g + f \frac{\partial g}{\partial x_i}.$$

Example

Let:

$$f(x) = x^2, \quad g(x) = 3x.$$

Then:

$$\frac{d}{dx}[f(x)g(x)] = (2x)(3x) + (x^2)(3) = 6x^2 + 3x^2 = 9x^2.$$

Properties

$$\begin{aligned}\frac{d}{dx}(fg) &= f'g + fg' \\ \frac{d}{dx}(fgh) &= f'gh + fg'h + fgh'\end{aligned}$$

Product rule is used heavily in backpropagation
Especially in: linear layers, elementwise ops, attention

Relationship to Chain Rule

If:

$$h(x) = f(x)g(x)$$

Then:

$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

This rule interacts with chain rule when functions are nested.

Implementation Principle

Numerical differentiation:

$$\frac{d}{dx}(f(x)g(x)) \approx \frac{f(x+h)g(x+h) - f(x)g(x)}{h}.$$

Programming Task

Implement the product rule numerically.

```
def E08(f, g, x, h=1e-6):
    """
    Computes derivative of f(x) * g(x)
    using numerical approximation.
    """

    # compute product at x
    fg = f(x) * g(x)

    # compute product at x+h
    fg_ph = f(x + h) * g(x + h)

    # derivative by finite difference
    return (fg_ph - fg) / h
```

93 T05 – E09 : Quotient Rule Law

Mathematical Definition

Let two differentiable functions:

$$f(x), \quad g(x).$$

For the quotient:

$$h(x) = \frac{f(x)}{g(x)},$$

the **Quotient Rule** states:

$$\frac{d}{dx} \left(\frac{f(x)}{g(x)} \right) = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}.$$

In multivariable form:

$$\frac{\partial}{\partial x_i} \left(\frac{f}{g} \right) = \frac{\frac{\partial f}{\partial x_i} g - f \frac{\partial g}{\partial x_i}}{g^2}.$$

Example

Let:

$$f(x) = x^2, \quad g(x) = x + 1.$$

Then:

$$\frac{d}{dx} \left(\frac{x^2}{x+1} \right) = \frac{2x(x+1) - x^2(1)}{(x+1)^2}.$$

Properties

$$\frac{d}{dx} \left(\frac{f}{g} \right) = \frac{f'g - fg'}{g^2}$$

$$\frac{d}{dx} \left(\frac{1}{g} \right) = -\frac{g'}{g^2}$$

Quotient Rule = Product Rule + Chain Rule

$$\frac{f}{g} = f \cdot (g^{-1})$$

Relationship to Neural Networks

Used in:

- normalization layers,
- softmax denominators,
- rational activation functions,
- gradient calculation for loss functions with division.

Implementation Principle

Numerical finite difference:

$$\frac{d}{dx} \left(\frac{f}{g} \right) \approx \frac{\frac{f(x+h)}{g(x+h)} - \frac{f(x)}{g(x)}}{h}.$$

Programming Task

Implement the quotient rule numerically.

```
def E09(f, g, x, h=1e-6):
    """
    Computes derivative of f(x) / g(x)
    using numerical approximation.
    """

    # value at x
    fx = f(x)
    gx = g(x)
    q = fx / gx

    # value at x+h
    fx_ph = f(x + h)
    gx_ph = g(x + h)
    q_ph = fx_ph / gx_ph

    # derivative approximation
    return (q_ph - q) / h
```

94 T05 – E10 : Power Rule Law

Mathematical Definition

For a function of the form:

$$f(x) = x^n,$$

the **Power Rule** states:

$$\frac{d}{dx} x^n = nx^{n-1}.$$

This holds for:

- integer powers,
- real powers,
- negative powers,
- fractional powers.

Generalized Power Rule

For:

$$f(x) = [g(x)]^n,$$

$$\frac{d}{dx} [g(x)]^n = n[g(x)]^{n-1} g'(x) \quad (\text{Chain Rule + Power Rule}).$$

Examples

$$\frac{d}{dx} (x^3) = 3x^2,$$

$$\frac{d}{dx} (x^{-2}) = -2x^{-3},$$

$$\frac{d}{dx} (\sqrt{x}) = \frac{1}{2}x^{-1/2}.$$

Properties

$$\frac{d}{dx} (x^n) = nx^{n-1}$$

$$\frac{d}{dx} (ax^n) = anx^{n-1}$$

$$\frac{d}{dx} (x^{n+m}) = (n+m)x^{n+m-1}$$

Relationship to Other Calculus Laws

The power rule is a special form of:

$$\frac{d}{dx} e^{n \ln x}$$

so:

$$\frac{d}{dx} x^n = \frac{d}{dx} e^{n \ln x} = nx^{n-1}.$$

Implementation Principle

Numerical derivative:

$$\frac{d}{dx} x^n \approx \frac{(x + h)^n - x^n}{h}.$$

Programming Task

Implement the power rule numerically for any real exponent n .

```
def E10(x, n, h=1e-6):
    """
    Computes derivative of x^n using finite differences.
    """

    fx = x**n
    fx_ph = (x + h)**n

    return (fx_ph - fx) / h
```

95 T05 – E11 : Exponential Derivative Law

Mathematical Definition

For the exponential function:

$$f(x) = e^x,$$

the derivative is:

$$\frac{d}{dx} e^x = e^x.$$

This is a unique function because it is its own derivative.

General Exponential Law

For a function of the form:

$$f(x) = e^{g(x)},$$

the derivative is:

$$\frac{d}{dx} e^{g(x)} = e^{g(x)} \cdot g'(x) \quad (\text{Chain Rule}).$$

Examples

$$\frac{d}{dx} (e^{3x}) = 3e^{3x},$$

$$\frac{d}{dx} (e^{x^2}) = e^{x^2} \cdot 2x,$$

$$\frac{d}{dx} (e^{\sin x}) = e^{\sin x} \cos x.$$

Properties

$$\frac{d}{dx}(e^x) = e^x$$

$$\frac{d}{dx}(ae^x) = ae^x$$

$$\frac{d}{dx}(e^{g(x)}) = e^{g(x)}g'(x)$$

$$e^{x+y} = e^x e^y$$

Relationship to Machine Learning

The exponential derivative is used in:

- Softmax: $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Attention scores: $e^{q^T k}$
- Loss functions: cross-entropy
- Normalizing flows
- Diffusion and score-based models

Implementation Principle

Numerical derivative:

$$\frac{d}{dx} e^{g(x)} \approx \frac{e^{g(x+h)} - e^{g(x)}}{h}.$$

Programming Task

Implement the exponential derivative numerically.

```
def E11(g, x, h=1e-6):
    """
    Computes derivative of exp(g(x)) using finite differences.

    g : function R -> R
    """
    fx = __import__("math").exp(g(x))
    fx_ph = __import__("math").exp(g(x + h))

    return (fx_ph - fx) / h
```

96 T05 – E12 : Logarithmic Derivative Law

Mathematical Definition

For the natural logarithm:

$$f(x) = \ln(x), \quad x > 0,$$

the derivative is:

$$\frac{d}{dx} \ln(x) = \frac{1}{x}.$$

General Logarithmic Derivative

For a function of the form:

$$f(x) = \ln(g(x)),$$

$$\frac{d}{dx} \ln(g(x)) = \frac{g'(x)}{g(x)} \quad (\text{Chain Rule}).$$

This is one of the most important derivative identities in calculus.

Examples

$$\frac{d}{dx} \ln(x^2 + 1) = \frac{2x}{x^2 + 1},$$

$$\frac{d}{dx} \ln(\sin x) = \frac{\cos x}{\sin x} = \cot x,$$

$$\frac{d}{dx} \ln(e^{3x}) = \frac{3e^{3x}}{e^{3x}} = 3.$$

Properties

$$\frac{d}{dx} \ln(x) = \frac{1}{x}$$

$$\frac{d}{dx} \ln(g(x)) = \frac{g'(x)}{g(x)}$$

$$\frac{d}{dx} \ln(fg) = \frac{f'}{f} + \frac{g'}{g}$$

$$\frac{d}{dx} \ln\left(\frac{f}{g}\right) = \frac{f'}{f} - \frac{g'}{g}$$

Relationship to Machine Learning

The logarithmic derivative appears in:

- Cross-entropy loss
- Log-softmax
- KL divergence
- Normalizing flows
- Log-likelihood training

For example, cross-entropy:

$$L = -\ln(\text{softmax}(x_i)).$$

$\frac{\partial L}{\partial x_i}$ uses the logarithmic derivative.

Implementation Principle

Numerical differentiation:

$$\frac{d}{dx} \ln(g(x)) \approx \frac{\ln(g(x+h)) - \ln(g(x))}{h}.$$

Programming Task

Write a Python function that computes the derivative of $\ln(g(x))$ numerically.

```
import math

def E12(g, x, h=1e-6):
    """
    Computes derivative of ln(g(x))
    using finite differences.
    """

    fx = math.log(g(x))
    fx_ph = math.log(g(x + h))

    return (fx_ph - fx) / h
```

97 T05 – E13 : Sigmoid Derivative Law

Mathematical Definition

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Its derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

This is one of the most elegant and important derivative identities in machine learning.

Derivation

Let:

$$\sigma(x) = (1 + e^{-x})^{-1}.$$

Using chain rule and power rule:

$$\sigma'(x) = -(1 + e^{-x})^{-2} \cdot (-e^{-x}) = \frac{e^{-x}}{(1 + e^{-x})^2}.$$

Now multiply numerator and denominator by e^x :

$$\sigma'(x) = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}} \right) = \sigma(x)(1 - \sigma(x)).$$

Examples

$$\sigma(0) = \frac{1}{2}, \quad \sigma'(0) = \frac{1}{4}.$$

$$\sigma(2) \approx 0.88, \quad \sigma'(2) = 0.88(1 - 0.88) \approx 0.1056.$$

Properties

$$0 < \sigma(x) < 1$$

$$0 < \sigma'(x) \leq 0.25$$

σ' is symmetric around $x = 0$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Relationship to Neural Networks

Sigmoid derivative is used in:

- Logistic regression gradient
- Binary cross-entropy loss
- Output layers for binary classification
- Activation backward pass in neural networks

The compact derivative makes backpropagation efficient.

Implementation Principle

Numerical approximation:

$$\sigma'(x) \approx \frac{\sigma(x + h) - \sigma(x)}{h}.$$

Programming Task

Write a Python function that computes the sigmoid derivative numerically.

```
import math

def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def E13(x, h=1e-6):
    """
    Computes derivative of sigmoid(x)
    using finite differences.
    """

    fx = sigmoid(x)
    fx_ph = sigmoid(x + h)

    return (fx_ph - fx) / h
```

98 T05 – E14 : Tanh Derivative Law

Mathematical Definition

The hyperbolic tangent function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

The derivative of $\tanh(x)$ is:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

This simple identity is widely used in neural networks.

Derivation

Start from the identity:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}.$$

Using quotient rule:

$$\frac{d}{dx} \tanh(x) = \frac{\cosh(x) \cosh(x) - \sinh(x) \sinh(x)}{\cosh^2(x)}.$$

Since:

$$\cosh^2(x) - \sinh^2(x) = 1,$$

$$\frac{d}{dx} \tanh(x) = \frac{1}{\cosh^2(x)} = \operatorname{sech}^2(x) = 1 - \tanh^2(x).$$

Examples

$$\tanh(0) = 0, \quad \tanh'(0) = 1.$$

$$\tanh(2) \approx 0.964, \quad \tanh'(2) \approx 1 - 0.964^2.$$

Properties

$$-1 < \tanh(x) < 1$$

$$0 < \tanh'(x) \leq 1$$

$\tanh(x)$ is symmetric around 0

$$\tanh'(x) = 1 - \tanh^2(x)$$

Relationship to Neural Networks

Used heavily in:

- RNN and LSTM activations
- Gated recurrent units (GRU)
- Encoder-decoder networks
- Normalization flows

Tanh is preferred over sigmoid in deeper networks because its derivative is larger near zero.

Implementation Principle

Numerical derivative of $\tanh(x)$:

$$\tanh'(x) \approx \frac{\tanh(x + h) - \tanh(x)}{h}.$$

Programming Task

Compute the derivative of $\tanh(x)$ numerically.

```
import math

def tanh(x):
    return math.tanh(x)

def E14(x, h=1e-6):
    """
    Numerical derivative of tanh(x)
    using finite differences.
    """

    fx = tanh(x)
    fx_ph = tanh(x + h)

    return (fx_ph - fx) / h
```

99 T05 – E15 : ReLU Derivative Law

Mathematical Definition

The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x).$$

Its derivative is:

$$\text{ReLU}'(x) = \begin{cases} 0 & x < 0, \\ 1 & x > 0, \\ \text{undefined or } 0/1 & x = 0. \end{cases}$$

Most deep learning libraries define:

$$\text{ReLU}'(0) = 0.$$

Behavior

$$\text{ReLU}(x) = \begin{cases} 0 & (\text{negative region}) \\ x & (\text{positive region}) \end{cases}$$

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{negative branch} \\ 1 & \text{positive branch} \end{cases}$$

Examples

$$\text{ReLU}(-3) = 0, \quad \text{ReLU}'(-3) = 0.$$

$$\text{ReLU}(4) = 4, \quad \text{ReLU}'(4) = 1.$$

Properties

ReLU is non-linear
ReLU derivative is piecewise constant
ReLU helps avoid vanishing gradients
 $\text{ReLU}'(x) = 0$ or 1
Not differentiable at $x = 0$

Relationship to Neural Networks

ReLU is widely used because:

- It is simple and fast.
- Its derivative does not saturate.
- It allows deeper networks without exploding/vanishing gradients.
- Works well with GPU tensor operations.

Implementation Principle

Use piecewise condition:

$$\text{ReLU}'(x) \approx \frac{\max(0, x+h) - \max(0, x)}{h}.$$

Programming Task

Write a Python function that computes the derivative of ReLU numerically.

```
def relu(x):
    return x if x > 0 else 0

def E15(x, h=1e-6):
    """
    Numerical derivative of ReLU(x)
    using finite differences.
    """

    fx = relu(x)
    fx_ph = relu(x + h)

    return (fx_ph - fx) / h
```

100 T05 – E16 : LeakyReLU Derivative Law

Mathematical Definition

The LeakyReLU function is defined as:

$$\text{LeakyReLU}(x) = \begin{cases} \alpha x & x < 0, \\ x & x \geq 0, \end{cases}$$

where $0 < \alpha < 1$ is a small slope (e.g., $\alpha = 0.01$).

The derivative is:

$$\text{LeakyReLU}'(x) = \begin{cases} \alpha & x < 0, \\ 1 & x \geq 0. \end{cases}$$

Behavior

$$\text{LeakyReLU}(x) = \begin{cases} \alpha x & \text{negative region} \\ x & \text{positive region} \end{cases}$$

$$\text{LeakyReLU}'(x) = \begin{cases} \alpha & \text{negative slope} \\ 1 & \text{positive slope} \end{cases}$$

Examples

For $\alpha = 0.01$:

$$\text{LeakyReLU}(-3) = -0.03, \quad \text{LeakyReLU}'(-3) = 0.01.$$

$$\text{LeakyReLU}(4) = 4, \quad \text{LeakyReLU}'(4) = 1.$$

Properties

- LeakyReLU avoids dead neurons
- Derivative is never zero
- Continuous but not differentiable at $x = 0$
- Slope in negative region is tunable (α)

Relationship to Neural Networks

LeakyReLU is useful in:

- deep networks suffering from dying ReLU problem
- GAN discriminator networks
- residual networks
- feature extraction layers

Implementation Principle

Numerical approximation:

$$\text{LeakyReLU}'(x) \approx \frac{\text{LeakyReLU}(x + h) - \text{LeakyReLU}(x)}{h}.$$

Programming Task

Implement LeakyReLU derivative numerically.

```
def leaky_relu(x, alpha=0.01):
    return x if x >= 0 else alpha * x

def E16(x, alpha=0.01, h=1e-6):
    """
    Numerical derivative of LeakyReLU(x)
    using finite differences.
    """

    fx = leaky_relu(x, alpha)
    fx_ph = leaky_relu(x + h, alpha)

    return (fx_ph - fx) / h
```

101 T05 – E17 : Softmax Derivative Law

Mathematical Definition

The softmax function for a vector

$$\mathbf{x} = [x_1, x_2, \dots, x_n]$$

is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}.$$

Let:

$$s_i = \text{softmax}(x_i).$$

The derivative of softmax is **not a scalar**, but a **Jacobian matrix**:

$$\frac{\partial s_i}{\partial x_j} = \begin{cases} s_i(1 - s_i), & i = j, \\ -s_i s_j, & i \neq j. \end{cases}$$

In matrix form:

$$J_{\text{softmax}}(\mathbf{x}) = \text{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^T.$$

This is one of the most important identities in deep learning.

Examples

If:

$$\mathbf{s} = [s_1, s_2],$$

then:

$$J = \begin{bmatrix} s_1(1 - s_1) & -s_1 s_2 \\ -s_2 s_1 & s_2(1 - s_2) \end{bmatrix}.$$

Properties

$$\sum_j \frac{\partial s_i}{\partial x_j} = 0 \quad (\text{probabilities sum to 1})$$

$$J = \text{diag}(\mathbf{s}) - \mathbf{s}\mathbf{s}^T \quad (\text{compact form})$$

Coupled derivatives (each output affects all others)

Used in softmax-crossentropy gradient

Relationship to Neural Networks

Softmax derivative is essential for:

- classifier output layers,
- transformer attention (scaled dot-product),
- cross-entropy gradient,
- RL policy-gradient training,
- probability distribution modeling.

Especially:

$$\frac{\partial}{\partial x_i} (-\ln s_k) = s_i - \mathbb{1}(i = k)$$

This is the core of **softmax + cross-entropy training**.

Implementation Principle

Numerical derivative:

$$\frac{\partial s_i}{\partial x_j} \approx \frac{\text{softmax}(x + h e_j)_i - \text{softmax}(x)_i}{h}.$$

Programming Task

Write a Python function that computes the full softmax Jacobian numerically.

```
import math

def softmax(xs):
    exps = [math.exp(x) for x in xs]
    s = sum(exps)
    return [e/s for e in exps]

def E17(xs, h=1e-6):
    """
    Computes the Jacobian of softmax(xs)
    using finite differences.

    xs : list of n values
    returns: n x n matrix
    """

    n = len(xs)
    s = softmax(xs)

    J = [[0.0 for _ in range(n)] for _ in range(n)]

    for j in range(n):
        xs_ph = xs.copy()
        xs_ph[j] += h

        s_ph = softmax(xs_ph)

        for i in range(n):
            J[i][j] = (s_ph[i] - s[i]) / h

    return J
```

102 T05 – E18 : Loss Gradient Law

Mathematical Definition

Let a model output:

$$\hat{y} = f(\mathbf{x}, \theta),$$

and a loss function:

$$L(\hat{y}, y),$$

where y is the target value and θ are model parameters.

The **loss gradient** with respect to parameters is given by:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \theta}.$$

This is the fundamental law of backpropagation.

For Scalar Output

If:

$$\hat{y} = f(\theta),$$

then:

$$\frac{dL}{d\theta} = L'(\hat{y}) \cdot f'(\theta).$$

For Vector Output

If:

$$\hat{\mathbf{y}} = f(\theta) \in \mathbb{R}^n,$$

then:

$$\frac{dL}{d\theta} = \sum_{i=1}^n \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial \theta}.$$

Examples

1. Mean Squared Error (MSE)

$$L = \frac{1}{2}(\hat{y} - y)^2$$

$$\frac{\partial L}{\partial \hat{y}} = (\hat{y} - y).$$

2. Binary Cross-Entropy

$$L = -[y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y})]$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}.$$

3. Softmax Cross-Entropy

$$L = -\ln(\hat{y}_k)$$

$$\frac{\partial L}{\partial \hat{y}_i} = \hat{y}_i - \mathbb{1}(i = k).$$

Properties

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \theta}$$

Backpropagation = repeated chain rule

Loss gradient drives optimization

Relationship to Neural Networks

All neural network training computes:

$$\nabla_{\theta} L$$

using:

- Forward pass: compute \hat{y}
- Backward pass: compute loss gradients
- Optimizer: update parameters using gradients

$$\theta_{\text{new}} = \theta - \eta \frac{\partial L}{\partial \theta}$$

Implementation Principle

Numerical approximation:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(f(\theta + h)) - L(f(\theta))}{h}.$$

Programming Task

Write a Python function that computes the loss gradient numerically.

```
def E18(L, f, theta, h=1e-6):
    """
    Computes dL/dtheta using numerical differentiation.

    L : loss function taking (y_pred, y_true)
    f : model function mapping theta -> prediction
    theta : current parameter value
    """

    # compute L(theta)
    y = f(theta)
    L0 = L(y)

    # compute L(theta + h)
    y_ph = f(theta + h)
    L1 = L(y_ph)

    # numerical gradient
    return (L1 - L0) / h
```

103 Optimization Laws

The purpose of **T06** is to introduce the mathematical foundations of **optimization methods** used in training neural networks. Every modern AI model—including CNNs, RNNs, Transformers, and LLMs—relies on these optimization laws to adjust parameters and minimize loss functions.

Optimization is the process of updating model parameters θ using gradients:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} L,$$

where η is the learning rate and L is the loss function.

In **T06**, each exercise introduces one optimization method, explains its mathematical update rule, and implements it in pure Python (without PyTorch) so the student understands how optimizers work internally.

This track includes the following laws:

- E00 – Gradient Descent Law
- E01 – Stochastic Gradient Descent Law
- E02 – Momentum Law
- E03 – RMSProp Law
- E04 – Adam Optimizer Law
- E05 – AdamW Law
- E06 – Nesterov Momentum Law
- E07 – Learning Rate Decay Law
- E08 – Weight Decay Law

By completing T06, the student will understand how optimization algorithms compute updates, how they handle momentum, adaptive learning rates, and regularization, and how they affect training stability and convergence in deep neural networks.

104 T06 – E00 : Gradient Descent Law

Mathematical Definition

Let a model parameter:

$$\theta \in \mathbb{R},$$

and a loss function:

$$L(\theta) : \mathbb{R} \rightarrow \mathbb{R}.$$

The **Gradient Descent** update rule is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{dL}{d\theta},$$

where:

- $\eta > 0$ is the learning rate,
- $\frac{dL}{d\theta}$ is the gradient of the loss.

This rule moves the parameter in the direction of steepest descent.

Vector Form

For vector parameters:

$$\theta \in \mathbb{R}^n,$$

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} L.$$

Example

Let:

$$L(\theta) = (\theta - 3)^2.$$

Gradient:

$$\frac{dL}{d\theta} = 2(\theta - 3).$$

Update:

$$\theta_{\text{new}} = \theta - 2\eta(\theta - 3).$$

Properties

GD converges if $0 < \eta < \frac{1}{L_{\max}}$

$\theta \rightarrow \theta^*$ where $\nabla L(\theta^*) = 0$

Too large η causes divergence

Too small η causes slow learning

Relationship to Neural Networks

Gradient Descent is the foundation of:

- SGD
- Momentum
- Adam
- RMSProp
- Adaptive optimizers

Every optimizer is a modification of basic gradient descent.

Implementation Principle

Numerical gradient:

$$\frac{dL}{d\theta} \approx \frac{L(\theta + h) - L(\theta)}{h}.$$

Programming Task

Implement one step of Gradient Descent.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E00(L, theta, eta):
    """
    Performs one step of Gradient Descent.

    L : loss function
    theta : current parameter
    eta : learning rate
    """

    g = numerical_grad(L, theta)
    theta_new = theta - eta * g
    return theta_new
```

105 – T06 – E01 : Stochastic Gradient Descent Law

Mathematical Definition

Given a dataset:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N,$$

and model parameters θ , we define the loss for a single sample:

$$L_i(\theta) = L(f(x_i, \theta), y_i).$$

The **Stochastic Gradient Descent (SGD)** update rule is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} L_i(\theta),$$

where:

- One data sample (or mini-batch) is used per update.
- η is the learning rate.

Mini-Batch SGD

For a mini-batch $B \subset \mathcal{D}$:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{1}{|B|} \sum_{i \in B} \nabla_{\theta} L_i(\theta).$$

This is the most widely used training method in deep learning.

Examples

If:

$$L_i(\theta) = (\theta - y_i)^2,$$

then:

$$\frac{d}{d\theta} L_i = 2(\theta - y_i).$$

SGD update:

$$\theta_{\text{new}} = \theta - 2\eta(\theta - y_i).$$

Properties

- Faster updates than full-batch GD
- More noise = better generalization
- Mini-batch size affects stability
- SGD is the base of all advanced optimizers

Relationship to Neural Networks

Used in:

- CNN training
- Transformer training
- RL policy gradient
- GANs
- Any model with large datasets

Deep learning uses **mini-batch SGD** by default.

Implementation Principle

Numerical gradient for one sample:

$$\frac{dL_i}{d\theta} \approx \frac{L_i(\theta + h) - L_i(\theta)}{h}.$$

Programming Task

Implement one SGD step using numerical gradients.

```
def numerical_grad(Li, theta, h=1e-6):
    return (Li(theta + h) - Li(theta)) / h

def E01(Li, theta, eta):
    """
    Performs one step of SGD on a single loss Li.

    Li : loss function for one data point
```

```

theta : current parameter
eta : learning rate
"""
g = numerical_grad(Li, theta)
theta_new = theta - eta * g
return theta_new

```

106 T06 – E02 : Momentum Law

Mathematical Definition

Standard Gradient Descent updates parameters using:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L(\theta_t).$$

Momentum adds an additional velocity term v_t that accumulates past gradients:

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla_{\theta} L(\theta_t),$$

$$\theta_{t+1} = \theta_t - \eta v_{t+1}.$$

Where:

- $0 < \beta < 1$ is the momentum factor (typically 0.9),
- v_t is the accumulated velocity,
- η is the learning rate.

Interpretation

Momentum behaves like a physical ball rolling down a hill:

- It speeds up in consistent gradient directions,
- It slows down when gradients oscillate,
- It smooths training and avoids getting stuck.

Example

If gradients oscillate like:

$$+3, -3, +3, -3, \dots$$

Plain SGD jumps side-to-side. Momentum integrates these into a stable direction.

Properties

Faster convergence in deep valleys Reduces oscillation across steep slopes Builds velocity over time $\beta \approx 0.9$ works well in practice
--

Relation to Neural Networks

Momentum is used in:

- CNN training
- RNN and LSTM optimization
- Transformer architectures
- GAN training

Many optimizers (Adam, Nesterov) are improvements of Momentum.

Implementation Principle

Numerical gradient:

$$\nabla_{\theta} L(\theta) \approx \frac{L(\theta + h) - L(\theta)}{h}.$$

Programming Task

Implement one step of Momentum optimization.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E02(L, theta, v, eta, beta):
    """
    Performs one Momentum update step.

    L : loss function
    theta : parameter
    v : current velocity
    eta : learning rate
    beta : momentum coefficient
    """

    # compute gradient
    g = numerical_grad(L, theta)

    # update velocity
    v_new = beta * v + (1 - beta) * g

    # update parameter
    theta_new = theta - eta * v_new

    return theta_new, v_new
```

107 T06 – E03 : RMSProp Law

Mathematical Definition

RMSProp is an adaptive optimization algorithm that scales the learning rate based on a moving average of squared gradients.

Given a parameter:

$$\theta_t,$$

and gradient:

$$g_t = \nabla_{\theta} L(\theta_t),$$

RMSProp maintains an exponential moving average of squared gradients:

$$s_{t+1} = \rho s_t + (1 - \rho)g_t^2,$$

where:

- s_t is the accumulated squared gradient,
- $0 < \rho < 1$ is the decay rate (typically 0.9),
- g_t^2 is the element-wise square.

The parameter update rule is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_{t+1}} + \epsilon} g_t,$$

where ϵ is a small constant for numerical stability.

Interpretation

RMSProp automatically adjusts the step size:

- large gradients \rightarrow smaller learning rate,
- small gradients \rightarrow larger learning rate,
- stabilizes training on noisy or curved loss surfaces.

Properties

Adaptive per-parameter learning rate
Suitable for RNNs and non-stationary problems
Less oscillation than SGD or Momentum
 $\rho \approx 0.9$ works well in practice

Relation to Neural Networks

RMSProp is widely used in:

- LSTM and GRU training,
- reinforcement learning (Deep Q-Networks),
- unstable optimization problems,
- deep networks with noisy gradients.

It is the parent method of Adam:

$$\text{Adam} = \text{Momentum} + \text{RMSProp}.$$

Implementation Principle

Numerical gradient approximation:

$$g_t \approx \frac{L(\theta_t + h) - L(\theta_t)}{h}.$$

Programming Task

Implement one RMSProp update step.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E03(L, theta, s, eta, rho, eps=1e-8):
    """
    Performs one RMSProp update step.

    L    : loss function
    theta : parameter
    s    : accumulated squared gradient
    eta  : learning rate
    rho  : decay rate
    eps  : small value for stability
    """

    # compute gradient
    g = numerical_grad(L, theta)

    # update squared gradient accumulator
    s_new = rho * s + (1 - rho) * (g * g)

    # update parameter
    theta_new = theta - (eta / ((s_new ** 0.5) + eps)) * g

    return theta_new, s_new
```

108 T06 – E04 : Adam Optimizer Law

Mathematical Definition

Adam (Adaptive Moment Estimation) combines:

- Momentum (first moment),
- RMSProp (second moment),
- Bias correction.

Given gradient at step t :

$$g_t = \nabla_{\theta} L(\theta_t),$$

Adam maintains two moving averages:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \end{aligned}$$

These are bias-corrected:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}. \end{aligned}$$

The parameter update rule is:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

Where:

- $\beta_1 \approx 0.9$ (momentum),
- $\beta_2 \approx 0.999$ (RMSProp),
- $\epsilon \approx 10^{-8}$ (numerical stability),
- η is the learning rate.

Interpretation

Adam adaptively adjusts step sizes per parameter:

- Momentum smooths gradient direction,
- RMSProp scales learning rate by gradient variance,
- Bias correction prevents early underestimation.

Properties

	Fast convergence
	Low memory cost
	Stable for noisy gradients
	Default optimizer for deep learning
	$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

Relation to Neural Networks

Adam is used to train:

- Transformers (GPT, LLaMA, DeepSeek)
- CNNs
- RNN/LSTM
- Autoencoders
- Diffusion models
- Reinforcement learning agents

Deep learning frameworks use Adam as the standard optimizer.

Implementation Principle

Numerical gradient:

$$g_t \approx \frac{L(\theta_t + h) - L(\theta_t)}{h}.$$

Programming Task

Implement one Adam update step.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E04(L, theta, m, v, t, eta,
        beta1=0.9, beta2=0.999, eps=1e-8):
    """
    Performs one Adam update step.
    
```

```

L    : loss function
theta : current parameter
m, v : first and second moment accumulators
t    : timestep (starting from 1)
eta  : learning rate
"""

# gradient
g = numerical_grad(L, theta)

# update moment estimates
m_new = beta1 * m + (1 - beta1) * g
v_new = beta2 * v + (1 - beta2) * (g * g)

# bias correction
m_hat = m_new / (1 - beta1**t)
v_hat = v_new / (1 - beta2**t)

# parameter update
theta_new = theta - eta * m_hat / ((v_hat ** 0.5) + eps)

return theta_new, m_new, v_new

```

109 T06 – E05 : AdamW Law

Mathematical Definition

AdamW is a corrected version of Adam that separates:

- adaptive gradient update,
- weight decay regularization.

This fixes the flaws of L2 regularization inside Adam.

Given gradient:

$$g_t = \nabla_{\theta} L(\theta_t),$$

AdamW uses the same moment estimates as Adam:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \end{aligned}$$

Bias corrections:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}. \end{aligned}$$

Decoupled Weight Decay

Unlike Adam, AdamW applies weight decay separately:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right),$$

where:

- λ is the weight decay coefficient,
- $\epsilon \approx 10^{-8}$,
- $\beta_1 = 0.9, \beta_2 = 0.999$.

Interpretation

AdamW fixes the incorrect L2 penalty in Adam by:

- applying weight decay directly to parameters,
- improving generalization,
- preventing parameter explosion,
- stabilizing training for large models.

Properties

Better generalization than Adam
Correct weight decay behavior
Used in all modern LLM training
 $\lambda \approx 0.01$ typical value

Relation to Neural Networks

AdamW is the optimizer used to train:

- GPT-2, GPT-3, GPT-4 architectures,
- LLaMA family,
- DeepSeek models,
- Vision Transformers (ViT),
- Diffusion models,
- Any foundation-model-scale architecture.

Implementation Principle

Numerical gradient approximation:

$$g_t \approx \frac{L(\theta_t + h) - L(\theta_t)}{h}.$$

Programming Task

Implement one AdamW update step.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E05(L, theta, m, v, t, eta,
        beta1=0.9, beta2=0.999,
        eps=1e-8, weight_decay=0.01):
    """
    Performs one AdamW update step.

    L : loss function
    theta : parameter
    m, v : moment accumulators
    t : timestep
    eta : learning rate
    weight_decay : lambda
    """
    pass
```

```

# gradient
g = numerical_grad(L, theta)

# update moments
m_new = beta1 * m + (1 - beta1) * g
v_new = beta2 * v + (1 - beta2) * (g * g)

# bias correction
m_hat = m_new / (1 - beta1**t)
v_hat = v_new / (1 - beta2**t)

# AdamW update
theta_new = theta \
    - eta * (m_hat / ((v_hat ** 0.5) + eps) \
    + weight_decay * theta)

return theta_new, m_new, v_new

```

110 T06 – E06 : Nesterov Momentum Law

Mathematical Definition

Nesterov Momentum is an improved version of classical momentum. Instead of computing the gradient at the current parameter θ_t , it computes the gradient at the *look-ahead* position:

$$\theta_t^{\text{look}} = \theta_t - \beta v_t.$$

Given gradient:

$$g_t = \nabla_{\theta} L(\theta_t^{\text{look}}),$$

the update rules are:

$$v_{t+1} = \beta v_t + \eta g_t,$$

$$\theta_{t+1} = \theta_t - v_{t+1}.$$

Where:

- v_t is the momentum velocity,
- β is the momentum factor (typically 0.9),
- η is the learning rate.

Interpretation

Nesterov Momentum "looks ahead" before making an update:

- anticipates the future position,
- corrects the overshoot of classical momentum,
- provides smoother and more accurate updates,
- improves performance in curved loss surfaces.

Comparison to Classical Momentum

Momentum:

$$v_{t+1} = \beta v_t + g_t,$$
$$\theta_{t+1} = \theta_t - \eta v_{t+1}.$$

Nesterov:

$$g_t = \nabla_\theta L(\theta_t - \beta v_t),$$
$$v_{t+1} = \beta v_t + \eta g_t.$$

Nesterov is more stable and often converges faster.

Properties

- Look-ahead update improves accuracy
- Less oscillation across steep slopes
- Often faster than classical momentum
- $\beta \approx 0.9$ commonly used

Relation to Neural Networks

Used in:

- CNNs,
- RNN training,
- deep reinforcement learning,
- large-scale optimization tasks.

Though Adam/AdamW are most common today, Nesterov still performs well in specific architectures.

Implementation Principle

Numerical gradient approximation:

$$g_t \approx \frac{L(\theta_t^{\text{look}} + h) - L(\theta_t^{\text{look}})}{h}.$$

Programming Task

Implement one Nesterov Momentum update step.

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E06(L, theta, v, eta, beta):
    """
    Performs one Nesterov Momentum update step.

    L : loss function
    theta : parameter
    v : velocity
    eta : learning rate
    beta : momentum coefficient
    """

    # look-ahead parameter
    theta_look = theta - beta * v
```

```

# gradient at look-ahead position
g = numerical_grad(L, theta_look)

# update velocity
v_new = beta * v + eta * g

# update parameter
theta_new = theta - v_new

return theta_new, v_new

```

111 T06 – E07 : Learning Rate Decay Law

Mathematical Definition

Learning rate decay reduces the learning rate over time to achieve more stable convergence during training.

Given initial learning rate:

$$\eta_0,$$

and step index t , the decayed learning rate η_t is computed depending on the schedule type.

1. Exponential Decay

$$\eta_t = \eta_0 \gamma^t,$$

where $0 < \gamma < 1$.

2. Step Decay

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor \frac{t}{k} \rfloor},$$

where k is the step interval.

3. Polynomial Decay

$$\eta_t = \eta_0 \left(1 - \frac{t}{T}\right)^p,$$

where T is total steps and p is power.

4. Cosine Decay (used in Transformers)

$$\eta_t = \frac{1}{2} \eta_0 \left(1 + \cos\left(\frac{\pi t}{T}\right)\right).$$

5. Warmup (LLM Training)

For $t < T_{\text{warmup}}$:

$$\eta_t = \eta_0 \frac{t}{T_{\text{warmup}}}.$$

Combined warmup + cosine:

$$\eta_t = \begin{cases} \eta_0 \frac{t}{T_{\text{warmup}}}, & t < T_{\text{warmup}}, \\ \frac{1}{2} \eta_0 \left(1 + \cos\left(\frac{\pi(t-T_{\text{warmup}})}{T-T_{\text{warmup}}}\right)\right), & t \geq T_{\text{warmup}}. \end{cases}$$

Interpretation

Learning rate decay helps:

- prevent divergence,
- stabilize late-stage training,
- reduce oscillation,
- improve final accuracy.

Relation to Neural Networks

Used in:

- GPT / LLaMA / DeepSeek training,
- CNNs and image classification,
- reinforcement learning agents,
- diffusion models.

Cosine + warmup is the standard for large-scale LLMs.

Programming Task

```
import math

def exponential_decay(eta0, gamma, t):
    return eta0 * (gamma ** t)

def step_decay(eta0, gamma, t, k):
    return eta0 * (gamma ** (t // k))

def polynomial_decay(eta0, t, T, p=2):
    return eta0 * ((1 - t / T) ** p)

def cosine_decay(eta0, t, T):
    return 0.5 * eta0 * (1 + math.cos(math.pi * t / T))

def warmup_cosine(eta0, t, T, Tw):
    if t < Tw:
        return eta0 * (t / Tw)
    return 0.5 * eta0 * (1 + math.cos(math.pi * (t - Tw) / (T - Tw)))
```

112 T06 – E08 : Weight Decay Law

Mathematical Definition

Weight decay is a regularization technique that penalizes large parameter values in order to prevent overfitting.

Given parameters:

$$\theta,$$

and a loss function:

$$L(\theta),$$

weight decay adds an additional penalty term:

$$L_{\text{total}} = L(\theta) + \frac{\lambda}{2} \|\theta\|^2.$$

The gradient becomes:

$$\nabla_{\theta} L_{\text{total}} = \nabla_{\theta} L(\theta) + \lambda \theta.$$

The update rule is therefore:

$$\theta_{t+1} = \theta_t - \eta (\nabla_{\theta} L(\theta_t) + \lambda \theta_t).$$

Where:

- $\lambda > 0$ is the weight decay factor,
- η is the learning rate.

Interpretation

Weight decay:

- shrinks weights toward zero,
- prevents overfitting,
- improves generalization,
- penalizes complex models.

It is equivalent to L2 regularization in classical machine learning.

Relationship to Optimizers

For SGD:

$$\theta_{t+1} = \theta_t - \eta (\nabla L + \lambda \theta_t).$$

For Adam:

L2 regularization inside Adam is incorrect.

Thus in modern deep learning:

Weight Decay \neq L2 Regularization in Adam.

This is why **AdamW** was created.

Properties

- Strong regularization
- Reduces model complexity
- Improves test accuracy
- $\lambda \in [10^{-5}, 10^{-2}]$ typical

Implementation Principle

Numerical gradient:

$$\nabla L(\theta) \approx \frac{L(\theta + h) - L(\theta)}{h}.$$

Total gradient:

$$g_{\text{total}} = g + \lambda \theta.$$

Programming Task

```
def numerical_grad(L, theta, h=1e-6):
    return (L(theta + h) - L(theta)) / h

def E08(L, theta, eta, weight_decay):
    """
    Performs one SGD step with weight decay.

    L : loss function
    theta : parameter
    eta : learning rate
    weight_decay : lambda
    """

    # compute gradient
    g = numerical_grad(L, theta)

    # apply weight decay to gradient
    g_total = g + weight_decay * theta

    # update parameter
    theta_new = theta - eta * g_total

    return theta_new
```

113 Statistics Probability Laws

The purpose of **T07** is to establish the fundamental statistical and probabilistic laws that form the mathematical basis of machine learning and artificial intelligence. Every concept in data science, from normalization to loss functions to probability models, relies on these core statistical definitions.

Statistics allows us to analyze data and measure its properties, while probability provides the mathematical framework for modeling uncertainty and randomness. Together, they form the foundation of all modern learning algorithms.

This track covers essential laws such as:

- E00 – Mean Law
- E01 – Median Law
- E02 – Mode Law
- E03 – Variance Law
- E04 – Standard Deviation Law
- E05 – Covariance Law
- E06 – Correlation Law
- E07 – Probability Density Function Law
- E08 – Normal Distribution Law
- E09 – Uniform Distribution Law
- E10 – Bernoulli Law
- E11 – Softmax Probability Law
- E12 – Entropy Law
- E13 – KL Divergence Law

- E14 – Jensen Inequality Law

By completing **T07**, the student will gain a strong understanding of the mathematical structure underlying data distributions, randomness, and information theory. These laws are essential for neural networks, loss functions, Bayesian methods, machine learning algorithms, and modern AI research.

114 T07 – E00 : Mean Law

Mathematical Definition

Given a set of n real numbers:

$$x = [x_1, x_2, \dots, x_n],$$

the **mean** (also called the average) is defined as:

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i.$$

Interpretation

The mean represents:

- the central value of a dataset,
- the expected value of a random variable,
- a measure of overall magnitude,
- the baseline used in normalization.

It is the most fundamental statistic in machine learning.

Properties

$$\text{Linear: } \text{mean}(ax + b) = a \cdot \text{mean}(x) + b$$

$$\text{Shift invariant: } \text{mean}(x + c) = \text{mean}(x) + c$$

Used in variance and probability models

Relation to Machine Learning

The mean is used in:

- normalization (zero-mean data),
- Gaussian distributions,
- batch normalization,
- loss functions (MSE, L1),
- statistical estimators.

Programming Task

Compute the mean of a list of values using pure Python.

```
def E00(xs):
    """
    Computes the mean of a list xs.
    """
    n = len(xs)
    return sum(xs) / n
```

115 T07 – E01 : Median Law

Mathematical Definition

Given a set of values:

$$x = [x_1, x_2, \dots, x_n],$$

let $x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n)}$ be the sorted values.

The **median** is defined as:

$$\text{median}(x) = \begin{cases} x_{(\frac{n+1}{2})}, & \text{if } n \text{ is odd,} \\ \frac{x_{(\frac{n}{2})} + x_{(\frac{n}{2}+1)}}{2}, & \text{if } n \text{ is even.} \end{cases}$$

Interpretation

The median represents the:

- central value of a dataset,
- point where half the data lies below and half above,
- robust estimator that is not affected by outliers.

Example:

Dataset: [1, 2, 100] Mean = 34.33 Median = 2 (correct center)

Properties

Insensitive to extreme values
Useful for skewed distributions
Used in robust statistics

Relation to Machine Learning

Median is used in:

- robust loss functions,
- median filtering in image processing,
- outlier detection methods,
- statistical feature engineering.

Programming Task

Compute the median of a list.

```
def E01(xs):
    """
    Computes the median of a list xs.
    """

    xs_sorted = sorted(xs)
    n = len(xs)

    if n % 2 == 1:
        # odd
```

```

    return xs_sorted[n // 2]
else:
    # even
    mid1 = xs_sorted[n // 2 - 1]
    mid2 = xs_sorted[n // 2]
    return (mid1 + mid2) / 2

```

116 T07 – E02 : Mode Law

Mathematical Definition

Given a dataset:

$$x = [x_1, x_2, \dots, x_n],$$

the **mode** is the value that appears most frequently.

Formally:

$$\text{mode}(x) = \arg \max_v |\{ i \mid x_i = v \}|.$$

Where:

$$|\{ i \mid x_i = v \}|$$

is the count of occurrences of value v .

Interpretation

Mode measures:

- the most common value,
- the dominant class,
- the peak of a discrete distribution.

Example:

$$x = [1, 4, 4, 5, 4, 2]$$

$$\text{mode}(x) = 4.$$

Properties

Useful for categorical data	
Can have multiple modes (multi-modal)	
	Unaffected by outliers

Relation to Machine Learning

Mode is used in:

- classification frequency analysis,
- majority voting algorithms,
- decision trees and random forests,
- cluster mode estimation.

Programming Task

Compute the mode of a list.

```
def E02(xs):
    """
    Computes the mode of a list xs.
    Returns the value with highest frequency.
    """

    counts = {}

    # count occurrences
    for v in xs:
        if v not in counts:
            counts[v] = 0
        counts[v] += 1

    # find key with max count
    mode_value = max(counts, key=counts.get)
    return mode_value
```

117 T07 – E03 : Variance Law

Mathematical Definition

Given a dataset:

$$x = [x_1, x_2, \dots, x_n],$$

and its mean:

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i,$$

the **variance** is defined as:

$$\text{Var}(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2.$$

This measures the spread of the data around the mean.

Alternative Form

Variance can also be computed using:

$$\text{Var}(x) = \left(\frac{1}{n} \sum_{i=1}^n x_i^2 \right) - \mu^2.$$

Interpretation

Variance represents:

- how spread out the data is,
- sensitivity of data to fluctuations,
- the squared deviation from the mean,
- the "energy" of variation in the dataset.

If:

$$\text{Var}(x) = 0 \Rightarrow \text{All values are identical.}$$

Properties

$$\begin{aligned}\text{Var}(x + c) &= \text{Var}(x) \\ \text{Var}(ax) &= a^2 \text{Var}(x) \\ \text{Var}(x) &= \text{E}[x^2] - (\text{E}[x])^2\end{aligned}$$

Relation to Machine Learning

Variance is used in:

- normalization and standardization,
- Gaussian (normal) distributions,
- PCA dimensionality reduction,
- Batch Normalization,
- loss function analysis,
- variance–bias tradeoff,
- noise and uncertainty modeling.

It is one of the core metrics of statistical learning.

Programming Task

Compute the variance using pure Python.

```
def E03(xs):  
    """  
        Computes the variance of a list xs.  
    """  
  
    n = len(xs)  
    mean = sum(xs) / n  
  
    # compute squared deviations  
    sq_dev = [(x - mean)**2 for x in xs]  
  
    return sum(sq_dev) / n
```

118 T07 – E04 : Standard Deviation Law

Mathematical Definition

Given a dataset:

$$x = [x_1, x_2, \dots, x_n],$$

with mean:

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i,$$

the **standard deviation** (SD) is defined as:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}.$$

It is the square root of variance:

$$\sigma = \sqrt{\text{Var}(x)}.$$

Interpretation

Standard deviation represents:

- how far values lie from the mean,
- the natural scale of variation in data,
- sensitivity to abnormal fluctuations,
- the most widely used statistical dispersion measure.

Examples:

$$\sigma = 0 \Rightarrow \text{all values identical.}$$

Properties

$$\begin{aligned}\sigma(x + c) &= \sigma(x) \\ \sigma(ax) &= |a| \sigma(x) \\ \sigma &= \sqrt{\text{E}[x^2] - (\text{E}[x])^2}\end{aligned}$$

Relation to Machine Learning

Standard deviation is used in:

- normalization and z-score scaling,
- Gaussian distributions,
- loss function stability,
- uncertainty estimation,
- variance–bias tradeoff,
- Batch Normalization,
- signal processing and noise modeling.

Programming Task

Compute the standard deviation of a list using pure Python.

```
import math

def E04(xs):
    """
    Computes the standard deviation of list xs.
    """

    n = len(xs)
    mean = sum(xs) / n

    # compute squared deviations
    sq_dev = [(x - mean)**2 for x in xs]
    variance = sum(sq_dev) / n

    return math.sqrt(variance)
```

119 T07 – E05 : Covariance Law

Mathematical Definition

Given two datasets:

$$x = [x_1, x_2, \dots, x_n], \quad y = [y_1, y_2, \dots, y_n],$$

with means:

$$\mu_x = \frac{1}{n} \sum_{i=1}^n x_i, \quad \mu_y = \frac{1}{n} \sum_{i=1}^n y_i,$$

the **covariance** between x and y is:

$$\text{Cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y).$$

Interpretation

Covariance measures how two variables change together:

- $\text{Cov} > 0$: variables increase together,
- $\text{Cov} < 0$: one increases while the other decreases,
- $\text{Cov} = 0$: no linear relationship.

Examples:

$$x = [1, 2, 3], \quad y = [2, 4, 6] \Rightarrow \text{Cov} > 0$$

$$x = [1, 2, 3], \quad y = [6, 4, 2] \Rightarrow \text{Cov} < 0$$

Properties

$$\text{Cov}(x, x) = \text{Var}(x)$$

$$\text{Cov}(x, y) = \text{Cov}(y, x)$$

$$\text{Cov}(ax, by) = ab \text{Cov}(x, y)$$

Relation to Machine Learning

Covariance is used in:

- PCA (principal component analysis),
- multivariate Gaussian distributions,
- correlation computation,
- feature selection and redundancy analysis,
- portfolio optimization (finance ML applications).

Programming Task

Compute covariance between two lists.

```
def E05(xs, ys):
    """
    Computes covariance between xs and ys.
    Both lists must be of equal length.
    """

    n = len(xs)
    mean_x = sum(xs) / n
    mean_y = sum(ys) / n

    # compute sum of (x - mean_x)*(y - mean_y)
    cov = 0.0
    for i in range(n):
        cov += (xs[i] - mean_x) * (ys[i] - mean_y)

    return cov / n
```

120 T07 – E06 : Correlation Law

Mathematical Definition

Given two datasets:

$$x = [x_1, x_2, \dots, x_n], \quad y = [y_1, y_2, \dots, y_n],$$

their covariance is:

$$\text{Cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y),$$

and their standard deviations:

$$\sigma_x = \sqrt{\text{Var}(x)}, \quad \sigma_y = \sqrt{\text{Var}(y)}.$$

The **correlation coefficient** is:

$$\rho_{x,y} = \frac{\text{Cov}(x, y)}{\sigma_x \sigma_y}.$$

Interpretation

Correlation measures the *strength and direction* of the linear relationship:

$$\begin{aligned} \rho = 1 &\Rightarrow \text{perfect positive correlation} \\ \rho = -1 &\Rightarrow \text{perfect negative correlation} \\ \rho = 0 &\Rightarrow \text{no linear relationship} \end{aligned}$$

Properties

$$\begin{aligned} -1 &\leq \rho \leq 1 \\ \rho(x, x) &= 1 \\ \rho(x, y) &= \rho(y, x) \\ \rho(ax, by) &= \text{sign}(ab) \rho(x, y) \end{aligned}$$

Relation to Machine Learning

Correlation is used in:

- feature selection,
- multivariate Gaussian models,
- PCA (principal component analysis),
- data preprocessing,
- anomaly detection,
- dimensionality reduction,
- attention similarity scores.

It is also the core concept behind cosine similarity:

$$\text{cosine}(x, y) = \frac{x \cdot y}{\|x\| \|y\|}.$$

Programming Task

Compute correlation between two lists using pure Python.

```
import math

def E06(xs, ys):
    """
    Computes the correlation coefficient between xs and ys.
    """

    n = len(xs)
    mean_x = sum(xs) / n
    mean_y = sum(ys) / n

    # compute covariance
    cov = 0.0
    for i in range(n):
        cov += (xs[i] - mean_x) * (ys[i] - mean_y)
    cov /= n

    # compute standard deviations
    var_x = sum((x - mean_x)**2 for x in xs) / n
    var_y = sum((y - mean_y)**2 for y in ys) / n

    sigma_x = math.sqrt(var_x)
    sigma_y = math.sqrt(var_y)

    return cov / (sigma_x * sigma_y)
```

121 T07 – E07 : Probability Density Function (PDF) Law

Mathematical Definition

A **probability density function (PDF)** is a function

$$f(x) \geq 0$$

such that:

$$\int_{-\infty}^{+\infty} f(x) dx = 1.$$

For any interval $[a, b]$:

$$P(a \leq X \leq b) = \int_a^b f(x) dx.$$

The PDF describes the likelihood of a continuous random variable.

Properties

$$\begin{aligned} f(x) &\geq 0 \\ \int_{-\infty}^{+\infty} f(x) dx &= 1 \\ P(X = x) &= 0 \quad \text{for continuous variables} \end{aligned}$$

The PDF is not a probability by itself; the area under the curve represents probability.

Example: Gaussian PDF

The most important PDF in AI is the normal distribution:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

This describes noise, uncertainty, and appears throughout machine learning.

Interpretation

PDF represents:

- how probability mass is distributed,
- where values of a random variable concentrate,
- the shape of uncertainty in data,
- smooth probability for continuous variables.

Relation to Machine Learning

PDF is fundamental for:

- Gaussian Mixture Models,
- Variational Autoencoders (VAEs),
- generative diffusion models,
- Bayesian neural networks,
- noise modeling,
- likelihood-based learning.

PDF directly appears in the loss functions of many models:

$$\log f(x) \quad (\text{log-likelihood}).$$

Programming Task

Compute the Gaussian PDF for a value x .

```
import math

def E07(x, mu, sigma):
    """
    Computes the Gaussian PDF of x with mean mu and
    standard deviation sigma.
    """
    coeff = 1 / (math.sqrt(2 * math.pi) * sigma)
    exponent = -((x - mu)**2) / (2 * sigma**2)

    return coeff * math.exp(exponent)
```

122 T07 – E08 : Normal Distribution Law

Mathematical Definition

A random variable X follows a **Normal (Gaussian) distribution** with mean μ and standard deviation σ if its probability density function is:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

This is written as:

$$X \sim \mathcal{N}(\mu, \sigma^2).$$

Interpretation

The normal distribution describes:

- natural random noise,
- measurement uncertainty,
- errors in predictions,
- central behavior of many real-world processes,
- the fundamental shape of probability in statistics.

The curve is symmetric around the mean μ :

$$\text{Mean} = \mu, \quad \text{Variance} = \sigma^2.$$

68–95–99.7 Rule

For a normal distribution:

$$\begin{aligned} P(\mu - \sigma \leq X \leq \mu + \sigma) &\approx 68\% \\ P(\mu - 2\sigma \leq X \leq \mu + 2\sigma) &\approx 95\% \\ P(\mu - 3\sigma \leq X \leq \mu + 3\sigma) &\approx 99.7\% \end{aligned}$$

This rule is fundamental in statistics and machine learning.

Properties

Symmetric around μ
Bell-shaped curve
Fully determined by μ and σ
Maximum entropy distribution for fixed mean and variance

Relation to Machine Learning

Gaussian distributions appear in:

- weight initialization of neural networks,
- Bayesian inference,
- Kalman filters,
- Variational Autoencoders (VAEs),
- diffusion models,
- noise models in training,
- Gaussian Mixture Models (GMM),
- statistical feature preprocessing.

Many loss functions are derived from the Gaussian log-likelihood:

$$-\log f(x) = \frac{(x - \mu)^2}{2\sigma^2} + \text{const.}$$

Programming Task

Compute a Gaussian sample and its PDF.

```
import math
import random

def gaussian_pdf(x, mu, sigma):
    coeff = 1 / (math.sqrt(2 * math.pi) * sigma)
    exponent = -((x - mu)**2) / (2 * sigma**2)
    return coeff * math.exp(exponent)

def random_normal(mu, sigma):
    """
    Generates a random number from N(mu, sigma^2)
    using Box-Muller transform.
    """
    u1 = random.random()
    u2 = random.random()

    z = math.sqrt(-2 * math.log(u1)) * math.cos(2 * math.pi * u2)
    return mu + sigma * z
```

123 T07 – E09 : Uniform Distribution Law

Mathematical Definition

A random variable X is uniformly distributed on the interval $[a, b]$ if its probability density function (PDF) is:

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b, \\ 0, & \text{otherwise.} \end{cases}$$

This distribution is written as:

$$X \sim \text{Uniform}(a, b).$$

Interpretation

Uniform distribution means:

- all values in $[a, b]$ have equal probability,
- no preference for any region,
- perfect randomness across the interval.

The mean and variance are:

$$\mathbb{E}[X] = \frac{a+b}{2},$$
$$\text{Var}(X) = \frac{(b-a)^2}{12}.$$

Properties

$$f(x) = \text{constant on } [a, b]$$

Max entropy on a bounded interval

$$P(a \leq X \leq b) = 1$$

Relation to Machine Learning

Used in:

- weight initialization (e.g., Xavier/Glorot init),
- random sampling for training data,
- reinforcement learning exploration,
- Monte Carlo simulations,
- dropout mask generation,
- synthetic data generation.

Uniform sampling is a fundamental building block in ML randomness.

Programming Task

Generate uniform random numbers and compute the PDF.

```
import random

def uniform_pdf(x, a, b):
    """
    PDF of the uniform distribution on [a, b]
    """
    if a <= x <= b:
        return 1.0 / (b - a)
    return 0.0

def random_uniform(a, b):
    """
    Generate a random number from Uniform(a, b)
    """
    return a + (b - a) * random.random()
```

124 T07 – E10 : Bernoulli Law

Mathematical Definition

A random variable X follows a **Bernoulli distribution** with probability p if:

$$P(X = 1) = p, \quad P(X = 0) = 1 - p,$$

where $0 \leq p \leq 1$.

This is written as:

$$X \sim \text{Bernoulli}(p).$$

Probability Mass Function (PMF)

$$f(x) = p^x(1-p)^{1-x}, \quad x \in \{0, 1\}.$$

Mean and Variance

$$\begin{aligned}\mathbb{E}[X] &= p, \\ \text{Var}(X) &= p(1-p).\end{aligned}$$

Interpretation

Bernoulli models:

- binary outcomes (success/failure),
- single trial events,
- probability of yes/no decisions,
- outcomes of classification tasks.

Examples:

$X = 1 \Rightarrow$ success with probability p .

$X = 0 \Rightarrow$ failure with probability $1 - p$.

Properties

$$X^2 = X \quad (\text{because } X \in \{0, 1\})$$

$$\mathbb{E}[X] = p$$

$$\text{Var}(X) = p(1 - p)$$

Relation to Machine Learning

Bernoulli is fundamental in:

- binary classification (labels 0/1),
- logistic regression,
- sigmoid activation:

$$p = \sigma(z)$$

- binary cross-entropy loss:

$$L = -[y \ln(p) + (1 - y) \ln(1 - p)],$$

- dropout:

$$\text{mask} \sim \text{Bernoulli}(p),$$

- reinforcement learning (0/1 rewards).

Bernoulli is one of the most used probability laws in neural networks.

Programming Task

Generate Bernoulli samples and compute its PMF.

```
import random

def bernoulli_pmf(x, p):
    """
    PMF of Bernoulli(p)
    x must be 0 or 1
    """
    if x not in (0, 1):
        return 0.0
    return (p**x) * ((1 - p)**(1 - x))

def random_bernoulli(p):
    """
    Generate a Bernoulli random sample (0 or 1)
    """
    return 1 if random.random() < p else 0
```

125 T07 – E11 : Softmax Probability Law

Mathematical Definition

Given a real-valued vector:

$$\mathbf{x} = [x_1, x_2, \dots, x_n],$$

the **softmax function** converts it into a probability distribution:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}.$$

This ensures:

$$p_i = \text{softmax}(x_i) \geq 0, \quad \sum_{i=1}^n p_i = 1.$$

Interpretation

Softmax represents:

- how likely each class is,
- a smooth version of the argmax function,
- a normalized probability distribution,
- the basis of token prediction in LLMs.

The largest values receive the highest probability.

Temperature Scaling

Softmax can be controlled using temperature T :

$$\text{softmax}_T(x_i) = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}}.$$

- $T < 1$: sharper distribution (more confident),
- $T > 1$: flatter distribution (more random).

Properties

$$\begin{aligned} p_i &> 0 \\ \sum_i p_i &= 1 \end{aligned}$$

Softmax is invariant to additive constants:

$$\text{softmax}(x_i) = \text{softmax}(x_i + c)$$

Relation to Machine Learning

Softmax is used in:

- multi-class classification networks,
- cross-entropy loss,
- attention mechanism (Transformers),
- reinforcement learning policy functions,
- probabilistic sampling in LLMs,
- Boltzmann exploration.
- diffusion models.

Softmax is the core operation that turns logits into probabilities.

Programming Task

Implement softmax using pure Python (no libraries).

```
import math

def E11(xs):
    """
    Computes softmax probability distribution of xs.
    """

    # for numerical stability, subtract max
    m = max(xs)
    exps = [math.exp(x - m) for x in xs]

    s = sum(exps)
    return [e / s for e in exps]
```

126 T07 – E12 : Entropy Law

Mathematical Definition

For a discrete probability distribution:

$$p = [p_1, p_2, \dots, p_n], \quad \sum_{i=1}^n p_i = 1,$$

the **entropy** is defined as:

$$H(p) = - \sum_{i=1}^n p_i \log p_i.$$

Entropy measures the uncertainty or randomness in the distribution.

Interpretation

Entropy represents:

- uncertainty in prediction,
- amount of "surprise",
- number of bits needed to encode an outcome,
- disorder or randomness in data,
- model confidence in classification.

Examples:

$$H([1, 0]) = 0 \quad (\text{no uncertainty})$$

$$H([0.5, 0.5]) = \log 2 \quad (\text{maximum uncertainty})$$

Properties

$$\begin{aligned} H(p) &\geq 0 \\ H(p) &\text{ max when } p_i = 1/n \\ H(p) &= 0 \text{ if one event has prob. 1} \\ H(pq) &= H(p) + H(q) \text{ if independent} \end{aligned}$$

Relation to Machine Learning

Entropy is fundamental in:

- cross-entropy loss:

$$L = - \sum_i y_i \log p_i,$$

- Transformer attention normalization,
- reinforcement learning exploration,
- information gain in decision trees,
- text compression and token prediction,
- variational inference and generative models.

High entropy = uncertain model Low entropy = confident model

Continuous Version

For a PDF $f(x)$:

$$H(f) = - \int f(x) \log f(x) dx.$$

Programming Task

Compute entropy of a probability distribution.

```
import math

def E12(ps):
    """
    Computes entropy of a probability distribution ps.
    ps must sum to 1 and contain values > 0.
    """

    H = 0.0
    for p in ps:
        H += p * math.log(p)
    return -H
```

127 T07 – E13 : KL Divergence Law

Mathematical Definition

Given two probability distributions:

$$P = [p_1, p_2, \dots, p_n], \quad Q = [q_1, q_2, \dots, q_n],$$

the **Kullback–Leibler Divergence** from P to Q is:

$$D_{\text{KL}}(P \| Q) = \sum_{i=1}^n p_i \log \left(\frac{p_i}{q_i} \right).$$

KL divergence measures how much information is lost when Q is used to approximate P .

Interpretation

KL divergence represents:

- how “different” two distributions are,
- cost of assuming Q when the true distribution is P ,
- extra bits needed if using Q instead of P ,
- asymmetry:

$$D_{\text{KL}}(P \parallel Q) \neq D_{\text{KL}}(Q \parallel P).$$

If $P = Q$, then:

$$D_{\text{KL}}(P \parallel Q) = 0.$$

Properties

$$\begin{aligned} D_{\text{KL}}(P \parallel Q) &\geq 0 \quad (\text{Gibbs' inequality}) \\ D_{\text{KL}}(P \parallel Q) &= 0 \text{ iff } P = Q \end{aligned}$$

D_{KL} is not symmetric

D_{KL} measures relative entropy

Relation to Machine Learning

KL Divergence appears in:

- Variational Autoencoders (VAE latent prior):

$$L_{\text{VAE}} = \text{Reconstruction} + D_{\text{KL}}.$$

- Reinforcement Learning (policy updates),
- Distillation between models (teacher \rightarrow student),
- Language modeling (comparing logits),
- Attention normalization,
- Bayesian inference,
- Diffusion generative models.

KL Divergence is one of the most fundamental losses in probabilistic AI.

Continuous Version

For probability densities:

$$D_{\text{KL}}(P \parallel Q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx.$$

Programming Task

Compute KL divergence in pure Python.

```
import math

def E13(P, Q):
    """
    Computes KL divergence D_KL(P || Q)
    P and Q must be probability distributions
    of the same length.
    """

    kl = 0.0
    for p, q in zip(P, Q):
        kl += p * math.log(p / q)
    return kl
```

128 T07 – E14 : Jensen's Inequality Law

Mathematical Definition

Let f be a **convex function** and let X be a random variable. Jensen's inequality states:

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)].$$

If f is **concave**, the inequality reverses:

$$f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)].$$

Interpretation

Jensen's inequality describes how convex functions interact with expectations.

It means:

- if f is convex, then:

$$f(\text{average}) \leq \text{average of } f.$$

- if f is concave, then:

$$f(\text{average}) \geq \text{average of } f.$$

This inequality forms the basis of many bounds and approximations in machine learning.

Examples

For convex $f(x) = x^2$:

$$(\mathbb{E}[X])^2 \leq \mathbb{E}[X^2].$$

For concave $f(x) = \log x$:

$$\log \mathbb{E}[X] \geq \mathbb{E}[\log X].$$

Relation to KL Divergence and Variational Inference

Jensen's inequality is used to derive:

$$\log p(x) \geq \mathbb{E}_q \left[\log \frac{p(x, z)}{q(z)} \right],$$

which forms the basis of:

- Evidence Lower Bound (ELBO),
- Variational Autoencoders (VAE),
- approximate posterior inference,
- many probabilistic generative models.

Without Jensen's inequality, VAEs would not exist.

Convexity Condition

A function is convex if:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y), \quad 0 \leq \lambda \leq 1.$$

Programming Task

Check Jensen's inequality numerically for a convex function.

```
import random

def f(x):
    # convex function
    return x * x

def E14():
    # choose two random points
    x = random.random()
    y = random.random()
    lam = random.random()

    left = f(lam * x + (1 - lam) * y)
    right = lam * f(x) + (1 - lam) * f(y)

    return left <= right # should be True
```

129 Neural Network Transformation Laws

The purpose of **T08** is to introduce the fundamental operations and transformations used in neural networks. This track bridges the gap between tensor mathematics (T00–T07) and practical deep learning implementations.

In **T08**, each exercise focuses on a single neural network transformation law. The student reads the law, understands its purpose in deep learning, and then implements it using Python and PyTorch.

This track includes the following exercises:

- **E00 – Linear Transformation Law**
- **E01 – Convolution Law**
- **E02 – Transposed Convolution Law**
- **E03 – Depthwise Convolution Law**
- **E04 – Softmax Activation Law**
- **E05 – Sigmoid Activation Law**
- **E06 – ReLU Activation Law**
- **E07 – GELU Activation Law**

- E08 – Swish Activation Law
- E09 – Cross Entropy Loss Law
- E10 – MSE Loss Law (Mean Squared Error)
- E11 – L1 Loss Law (Mean Absolute Error)
- E12 – Binary Cross Entropy (BCE) Loss Law
- E13 – LogSoftmax Law
- E14 – LogSoftmax Gradient Law

By completing **T08**, the student will have a solid foundation to understand and implement neural networks, preparing them for advanced AI model design and experimentation.

130 T08 – E00 : Linear Transformation Law

Mathematical Definition

A **linear transformation** is a function defined as:

$$y = Wx + b,$$

where:

$$W \in \mathbb{R}^{m \times n}, \quad x \in \mathbb{R}^n, \quad b \in \mathbb{R}^m, \quad y \in \mathbb{R}^m.$$

This operation maps an input vector x into a new space using a weight matrix W and bias vector b .

Interpretation

The linear transformation performs:

- a scaling of the input,
- a rotation of the input space,
- a translation using the bias,
- projection into another dimension.

This operation is the fundamental building block of all neural networks.

Properties

A transformation is linear if:

$$T(\alpha x + \beta y) = \alpha T(x) + \beta T(y).$$

Bias b introduces an affine shift:

$$y = Wx + b \quad (\text{affine transformation}).$$

Relation to Machine Learning

Linear transformations form the core of:

- fully connected layers,
- embeddings in transformers,
- attention Q/K/V projections,
- convolution layers (locally linear),
- recurrent neural networks,
- autoencoders,
- generative models.

Every neural network block begins with:

$$\text{Linear}(x) = Wx + b.$$

Programming Task

Implement the linear transformation in pure Python.

```
def E00(x, W, b):
    """
    Computes y = Wx + b
    x: vector of length n
    W: matrix m x n
    b: vector of length m
    """

    m = len(W)
    n = len(W[0])

    y = [0] * m

    for i in range(m):
        s = 0
        for j in range(n):
            s += W[i][j] * x[j]
        y[i] = s + b[i]

    return y
```

131 T08 – E01 : Convolution Law

Mathematical Definition

The **1D convolution** of an input signal x and a kernel w is defined as:

$$y[n] = \sum_{k=0}^{K-1} x[n+k] w[k].$$

For 2D convolution (used in images):

$$Y(i, j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(i+m, j+n) W(m, n).$$

This sliding window operation multiplies and sums local regions of the input.

Interpretation

Convolution performs:

- feature extraction,
- edge detection,
- smoothing and blurring,
- sharpening,
- local pattern recognition,
- spatial filtering.

Each kernel W extracts a different kind of feature.

Properties

Locality: kernel sees only a small window

Weight sharing: W reused across the input

Equivariance: shift in input \rightarrow shift in output

Linear operation: convolution is linear

Relation to Machine Learning

Convolution is core to:

- CNN image models,
- audio/speech processing,
- time-series networks,
- UNet models,
- ResNet/ConvNeXt,
- modern vision transformers (patch embeddings),
- diffusion generative models.

It is one of the most fundamental operations in neural networks.

Programming Task

Implement a 1D convolution in pure Python.

```
def E01(x, w):
    """
    Performs 1D convolution without padding and without stride.
    x: input list of length N
    w: kernel list of length K
    returns output list of length N-K+1
    """

    N = len(x)
    K = len(w)
    y = []

    for i in range(N - K + 1):
        s = 0
```

```

for k in range(K):
    s += x[i + k] * w[k]
y.append(s)

return y

```

132 T08 – E02 : Transposed Convolution Law

Mathematical Definition

A **transposed convolution** is the mathematical inverse of a standard convolution in terms of spatial size. Given an input x and a kernel w , the transposed convolution expands the output size instead of reducing it.

For 1D transposed convolution:

$$y[n] = \sum_{k=0}^{K-1} x[k] w[n-k].$$

This operation increases the length of the signal.

For 2D (image) transposed convolution:

$$Y(i, j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(m, n) W(i - m, j - n).$$

This operation **upsamples** the spatial resolution.

Interpretation

Transposed convolution performs:

- upsampling,
- resolution expansion,
- image reconstruction,
- generating higher-resolution data,
- decoding latent representations.

It is commonly used to “undo” the downscaling effect of standard convolution.

Properties

Increases spatial size	
Kernel is flipped in operation	
Used in generative and decoder architectures	
Not a true mathematical inverse but shape inverse	

Relation to Machine Learning

Transposed convolution is fundamental in:

- GAN generators (DCGAN, StyleGAN),
- UNet decoders,
- super-resolution networks,
- semantic segmentation,

- autoencoder reconstruction,
- diffusion model upsampling layers.

Without transposed convolution, deep generative models cannot increase output resolution.

Programming Task

Implement a simple 1D transposed convolution in pure Python.

```
def E02(x, w):
    """
    Performs a simple 1D transposed convolution.
    x: input list of length N
    w: kernel list of length K
    returns output list of length N + K - 1
    """

    N = len(x)
    K = len(w)
    y = [0] * (N + K - 1)

    for i in range(N):
        for k in range(K):
            y[i + k] += x[i] * w[k]

    return y
```

133 T08 – E03 : Depthwise Convolution Law

Mathematical Definition

Depthwise convolution applies a separate convolution kernel to each input channel independently.

Given an input tensor:

$$X \in \mathbb{R}^{H \times W \times C},$$

and depthwise kernels:

$$W \in \mathbb{R}^{K \times K \times C},$$

the depthwise convolution output is:

$$Y(i, j, c) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} X(i+m, j+n, c) W(m, n, c).$$

Each channel is convolved independently without mixing.

Interpretation

Depthwise convolution performs:

- per-channel filtering,
- extremely efficient feature extraction,
- spatial filtering for each feature map,
- reduced computational cost compared to standard convolution.

It is the first step of **depthwise-separable convolution**.

Properties

No mixing between channels
Computational cost reduced by factor $1/C$
Used for lightweight neural networks
Spatial convolution only

Relation to Machine Learning

Depthwise convolution is fundamental in:

- MobileNet (DW + PW convolution),
- EfficientNet and EdgeTPU models,
- lightweight CNN architectures,
- mobile and embedded deep learning,
- early convolution stages in ViT,
- high-resolution feature extraction.

It dramatically reduces FLOPs while preserving accuracy.

Programming Task

Implement a simple depthwise convolution (1D version) in pure Python.

```
def E03(X, W):
    """
    Depthwise convolution (1D version).
    X: list of C channels, each a list of length N
    W: list of C kernels, each a list of length K

    Returns list of C outputs.
    """

    C = len(X)
    K = len(W[0])
    outputs = []

    for c in range(C):
        x = X[c]
        w = W[c]
        N = len(x)

        y = []
        for i in range(N - K + 1):
            s = 0
            for k in range(K):
                s += x[i + k] * w[k]
            y.append(s)

        outputs.append(y)

    return outputs
```

134 T08 – E04 : Softmax Activation Law

Mathematical Definition

Given a vector of logits:

$$z = [z_1, z_2, \dots, z_n],$$

the **softmax activation** converts logits into class probabilities:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}.$$

Softmax is differentiable, enabling backpropagation.

Interpretation

In neural networks, softmax activation:

- converts raw scores into probabilities,
- selects the most likely class,
- is used in output layers for classification,
- ensures numerical stability with log-sum-exp trick.

Properties

$$\begin{aligned}\sigma(z_i) &> 0, \\ \sum_i \sigma(z_i) &= 1, \\ \sigma(z + c) &= \sigma(z), \quad (\text{shift invariance}).\end{aligned}$$

Relation to Machine Learning

Softmax activation is fundamental in:

- multi-class neural networks,
- cross-entropy loss,
- Transformers (query–key attention weights),
- probabilistic sampling in LLMs,
- reinforcement learning policy networks.

Softmax is the final step that turns logits into a probability distribution.

Gradient

The derivative of softmax is:

$$\frac{\partial \sigma_i}{\partial z_j} = \sigma_i(\delta_{ij} - \sigma_j),$$

which forms the Jacobian matrix.

Programming Task

Implement softmax activation in pure Python.

```
import math

def E04(z):
    """
    Softmax activation for neural networks.
    z: list of logits
    """

    # numerical stability
    m = max(z)
    exps = [math.exp(v - m) for v in z]

    s = sum(exps)
    return [e / s for e in exps]
```

135 T08 – E05 : Sigmoid Activation Law

Mathematical Definition

The **sigmoid activation function** is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

It maps any real number to the interval $(0, 1)$.

Interpretation

Sigmoid activation:

- squashes values into the range $(0, 1)$,
- represents probability of binary events,
- is used in logistic regression,
- is used in LSTM gates and Bernoulli models.

$x \rightarrow \sigma(x)$ acts like a probability.

Properties

$$\begin{aligned} 0 < \sigma(x) < 1 \\ \sigma(-x) &= 1 - \sigma(x) \\ \lim_{x \rightarrow +\infty} \sigma(x) &= 1 \\ \lim_{x \rightarrow -\infty} \sigma(x) &= 0 \end{aligned}$$

Derivative

The derivative is crucial for backpropagation:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

This appears in every logistic regression and binary classification model.

Relation to Machine Learning

Sigmoid is used in:

- binary classification outputs,
- logistic regression,
- BCE loss (binary cross-entropy),
- reinforcement learning policies,
- LSTM input/forget/output gates,
- generative Bernoulli models.

It is the standard activation for probability modeling.

Programming Task

Implement the sigmoid activation in pure Python.

```
import math

def E05(x):
    """
    Computes the sigmoid activation.
    """
    return 1 / (1 + math.exp(-x))
```

136 T08 – E06 : ReLU Activation Law

Mathematical Definition

The **Rectified Linear Unit (ReLU)** is defined as:

$$\text{ReLU}(x) = \max(0, x).$$

It outputs x if $x > 0$, and 0 otherwise.

Interpretation

ReLU performs:

- non-linear activation,
- passing only positive signals,
- zeroing out negative values,
- introducing sparsity in activations.

$$x < 0 \Rightarrow 0, \quad x > 0 \Rightarrow x.$$

Properties

$$\text{ReLU}(x) = 0 \quad \text{if } x \leq 0$$

$$\text{ReLU}(x) = x \quad \text{if } x > 0$$

$$\text{Range: } [0, +\infty)$$

Non-linear: enables deep learning

Derivative

Crucial for backpropagation:

$$\frac{d}{dx} \text{ReLU}(x) = \begin{cases} 0, & x \leq 0, \\ 1, & x > 0. \end{cases}$$

Relation to Machine Learning

ReLU is the default activation in:

- CNN feature extractors,
- MLP layers,
- ResNet residual blocks,
- Transformers feed-forward blocks (FFN),
- GAN discriminators,
- high-dimensional deep networks.

It solves the vanishing gradient problem found in sigmoid and tanh.

Programming Task

Implement ReLU activation in pure Python.

```
def E06(x):
    """
    Computes the ReLU activation.
    """
    return x if x > 0 else 0
```

137 T08 – E07 : GELU Activation Law

Mathematical Definition

The **Gaussian Error Linear Unit (GELU)** is defined as:

$$\text{GELU}(x) = x \cdot \Phi(x),$$

where $\Phi(x)$ is the CDF of the standard normal distribution:

$$\Phi(x) = \frac{1}{2} \left[1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right].$$

An equivalent approximate formula used in Transformers:

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3) \right) \right).$$

Interpretation

GELU activation:

- smoothly gates negative values (not hard cutoff like ReLU),
- retains useful negative information,
- behaves probabilistically based on Gaussian curve,
- improves performance on deep transformers.

GELU "probabilistic ReLU".

Properties

Smooth and continuous
Keeps small negative values
Better than ReLU in Transformers
Non-linear and differentiable everywhere

Relation to Machine Learning

GELU is the default activation in:

- GPT models (all versions),
- BERT,
- RoBERTa,
- Vision Transformers (ViT),
- Stable Diffusion networks,
- large-scale deep learning systems.

It outperforms ReLU and Swish in many NLP and vision tasks.

Programming Task

Implement approximate GELU using pure Python.

```
import math

def E07(x):
    """
    Approximate GELU activation used in Transformers.
    """
    return 0.5 * x * (1 + math.tanh(
        math.sqrt(2 / math.pi) * (x + 0.044715 * x**3)
    ))
```

138 T08 – E08 : Swish Activation Law

Mathematical Definition

The **Swish activation function** is defined as:

$$\text{Swish}(x) = x \cdot \sigma(x),$$

where $\sigma(x)$ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Interpretation

Swish activation:

- allows small negative values (unlike ReLU),
- is smooth and non-monotonic,
- provides better gradient flow,

- enhances deep network performance,
- behaves similarly to GELU but is simpler.

Swish = “ReLU with intelligence”.

Properties

Smooth and differentiable
Non-zero output for negative inputs
Non-monotonic activation (beneficial for deep models)
Better performance than ReLU on many benchmarks

Derivative

Needed for gradient-based optimization:

$$\frac{d}{dx} \text{Swish}(x) = \sigma(x) + x\sigma(x)(1 - \sigma(x)).$$

Relation to Machine Learning

Swish is widely used in:

- EfficientNet architectures,
- MobileNetV3,
- Transformer feed-forward blocks (alternative to GELU),
- reinforcement learning policies,
- image classification and vision transformers,
- deep fully connected networks.

It often outperforms ReLU in accuracy and stability.

Programming Task

Implement Swish activation in pure Python.

```
import math

def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def E08(x):
    """
    Computes the Swish activation: x * sigmoid(x)
    """
    return x * sigmoid(x)
```

139 T08 – E09 : Cross Entropy Loss Law

Mathematical Definition

For a true distribution (ground truth):

$$y = [y_1, y_2, \dots, y_n],$$

and predicted probabilities (softmax output):

$$p = [p_1, p_2, \dots, p_n],$$

the **cross entropy loss** is defined as:

$$L(y, p) = - \sum_{i=1}^n y_i \log(p_i).$$

For one-hot encoded labels (classification):

$$y_k = 1 \Rightarrow L = -\log(p_k).$$

Interpretation

Cross entropy measures:

- how different the predicted distribution is from the true distribution,
- the cost of making wrong predictions,
- the uncertainty penalty,
- the negative log-likelihood of the correct class.

If the model is confident and correct:

$$p_k \approx 1 \Rightarrow L \approx 0.$$

If the model is wrong:

$$p_k \rightarrow 0 \Rightarrow L \rightarrow +\infty.$$

Binary Version

For binary classification ($y \in \{0, 1\}$):

$$L = -[y \log(p) + (1 - y) \log(1 - p)].$$

This is the **BCE loss**.

Relation to Machine Learning

Cross entropy loss is the fundamental loss for:

- image classification,
- language modeling (predicting tokens),
- GPT/LLM training,
- attention distributions,
- diffusion model likelihoods,
- reinforcement learning policy gradients,
- softmax-based networks.

Every LLM output token uses cross entropy.

Gradient

The derivative is simple and powerful:

$$\frac{\partial L}{\partial z_i} = p_i - y_i,$$

where z_i are logits.

This makes backpropagation stable and efficient.

Programming Task

Implement cross entropy loss in pure Python.

```
import math

def E09(y, p):
    """
    Computes cross entropy loss.
    y: one-hot vector
    p: predicted probabilities (softmax output)
    """

    loss = 0.0
    for yi, pi in zip(y, p):
        if yi == 1:
            loss = -math.log(pi)
            break
    return loss
```

140 T08 – E10 : MSE Loss Law

Mathematical Definition

Given predicted values:

$$\hat{y} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n],$$

and target (true) values:

$$y = [y_1, y_2, \dots, y_n],$$

the **Mean Squared Error (MSE)** is defined as:

$$L_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2.$$

Interpretation

MSE measures:

- how far predictions are from real values,
- the average squared difference,
- sensitivity to large errors,
- error magnitude for regression tasks.

If predictions are perfect:

$$\hat{y}_i = y_i \Rightarrow L = 0.$$

Large mistakes produce large penalty due to the square.

Properties

$L_{\text{MSE}} \geq 0$
 $L_{\text{MSE}} = 0$ iff $\hat{y}_i = y_i$
Convex loss function
Easy gradient computation

Gradient

The derivative used for backpropagation:

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{2}{n}(\hat{y}_i - y_i).$$

Relation to Machine Learning

MSE is used in:

- regression neural networks,
- forecasting models,
- autoencoders (reconstruction error),
- GAN generators (certain variants),
- reinforcement learning value functions,
- signal processing and denoising networks.

It is the most common loss for tasks involving real numbers.

Programming Task

Implement MSE loss in pure Python.

```
def E10(y, y_hat):
    """
    Computes Mean Squared Error loss.
    y: list of true values
    y_hat: list of predicted values
    """

    n = len(y)
    total = 0.0

    for yi, yhi in zip(y, y_hat):
        total += (yhi - yi) ** 2

    return total / n
```

141 T08 – E11 : L1 Loss Law

Mathematical Definition

Given predicted values:

$$\hat{y} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n],$$

and true values:

$$y = [y_1, y_2, \dots, y_n],$$

the **L1 Loss** or **Mean Absolute Error (MAE)** is:

$$L_{\text{L1}} = \frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i|.$$

Interpretation

L1 Loss measures:

- the average absolute difference,
- robustness to outliers (unlike MSE),
- sparsity in gradients,
- stable training for noisy datasets.

If predictions are perfect:

$$\hat{y}_i = y_i \Rightarrow L_{\text{L1}} = 0.$$

Properties

$$L_{\text{L1}} \geq 0$$

$$L_{\text{L1}} = 0 \text{ iff } \hat{y}_i = y_i$$

Less sensitive to outliers than MSE

Leads to sparse gradients

Gradient

The gradient used for backpropagation:

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{1}{n} \cdot \text{sign}(\hat{y}_i - y_i),$$

where

$$\text{sign}(x) = \begin{cases} +1, & x > 0, \\ -1, & x < 0, \\ 0, & x = 0. \end{cases}$$

Relation to Machine Learning

L1 Loss is used in:

- robust regression,
- Lasso regularization,
- GAN training stability,
- autoencoder reconstruction loss,
- computer vision tasks with outliers,
- reinforcement learning value estimation.

Programming Task

Implement L1 Loss in pure Python.

```
def E11(y, y_hat):
    """
    Computes L1 Loss (MAE).
    y: true values
    y_hat: predicted values
    """

    n = len(y)
    total = 0.0

    for yi, yhi in zip(y, y_hat):
        total += abs(yhi - yi)

    return total / n
```

142 T08 – E12 : Binary Cross Entropy (BCE) Loss Law

Mathematical Definition

For binary classification, with label:

$$y \in \{0, 1\},$$

and predicted probability:

$$p = \sigma(z) \in (0, 1),$$

the **Binary Cross Entropy (BCE) loss** is:

$$L_{\text{BCE}} = -[y \log(p) + (1 - y) \log(1 - p)].$$

Interpretation

BCE loss measures:

- how well the model predicts a binary outcome,
- penalty for confidence in the wrong direction,
- log-likelihood of a Bernoulli distribution,
- uncertainty of binary decisions.

If the model is correct with high probability:

$$y = 1, p \approx 1 \Rightarrow L \approx 0.$$

If the model is wrong and confident:

$$y = 1, p \rightarrow 0 \Rightarrow L \rightarrow +\infty.$$

Properties

$$\begin{aligned} L_{\text{BCE}} &\geq 0 \\ L_{\text{BCE}} &= 0 \text{ iff } p = y = 1 \end{aligned}$$

Convex in logistic regression

Stable with sigmoid activation

Relation to Machine Learning

BCE loss is used in:

- binary classifiers,
- GAN discriminator objective,
- Bernoulli output models,
- VAE binary decoders,
- reinforcement learning policies,
- logistic regression training.

Gradient

The gradient for backpropagation is:

$$\frac{\partial L}{\partial z} = p - y,$$

where:

$$p = \sigma(z).$$

Programming Task

Implement BCE loss in pure Python.

```
import math

def E12(y, p):
    """
    Computes Binary Cross Entropy loss.
    y: 0 or 1 (true label)
    p: predicted probability (sigmoid output)
    """
    return -(y * math.log(p) + (1 - y) * math.log(1 - p))
```

143 T08 – E13 : LogSoftmax Law

Mathematical Definition

Given logits:

$$z = [z_1, z_2, \dots, z_n],$$

the **log-softmax** is defined as:

$$\log \sigma(z_i) = z_i - \log \left(\sum_{j=1}^n e^{z_j} \right).$$

This is equivalent to:

$$\log \sigma(z_i) = \log(e^{z_i}) - \log \left(\sum_j e^{z_j} \right).$$

It produces log-probabilities.

Interpretation

LogSoftmax:

- converts logits into log-probabilities,
- avoids numerical overflow from large exponentials,
- is used internally in cross-entropy calculations,
- stabilizes gradients for deep models.

Properties

$$\sum_i e^{\log \sigma(z_i)} = 1$$
$$\log \sigma(z_i) \leq 0$$

More stable than softmax

Used directly in NLLLoss

Relation to Machine Learning

LogSoftmax is used in:

- PyTorch `nn.LogSoftmax`,
- `nn.NLLLoss` (negative log likelihood loss),
- transformer token prediction,
- GPT logits \rightarrow log probs,
- reinforcement learning log policies,
- stable cross-entropy implementations.

The combination:

LogSoftmax + NLLLoss

is the same as `**CrossEntropyLoss**`, but more numerically stable.

Programming Task

Implement log-softmax in pure Python.

```
import math

def E13(z):
    """
    Computes log-softmax of logits z.
    """

    # numerical stability: subtract max
    m = max(z)
    shifted = [v - m for v in z]

    # denominator = log(sum(exp(shifted)))
    denom = math.log(sum(math.exp(v) for v in shifted))

    # log-softmax for each element
    return [v - denom for v in shifted]
```

144 T08 – E14 : LogSoftmax Gradient Law

Mathematical Definition

Given logits:

$$z = [z_1, z_2, \dots, z_n],$$

and log-softmax outputs:

$$\ell_i = \log \sigma(z_i),$$

the gradient of the log-softmax with respect to the logits is:

$$\frac{\partial \ell_i}{\partial z_j} = \delta_{ij} - \sigma(z_j),$$

where:

$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Thus:

$$\frac{\partial \ell_i}{\partial z_i} = 1 - \sigma(z_i),$$

and for $i \neq j$:

$$\frac{\partial \ell_i}{\partial z_j} = -\sigma(z_j).$$

Interpretation

This means:

- increasing logit z_i increases its own log-probability,
- but decreases the others,
- gradient distributes influence across all classes,
- stable and smooth behavior for optimization.

The gradient matrix is a full Jacobian:

$$J_{ij} = \frac{\partial(\log \sigma(z_i))}{\partial z_j}.$$

Relation to Machine Learning

LogSoftmax gradient is used in:

- backpropagation through softmax outputs,
- GPT logits \rightarrow log-prob \rightarrow loss,
- transformer classifier heads,
- NLLLoss backward computation,
- reinforcement learning log-policy gradients,
- diffusion model denoising steps.

Together with NLLLoss:

$$\frac{\partial L}{\partial z_i} = p_i - y_i,$$

where $p_i = \sigma(z_i)$.

This is extremely stable for deep networks.

Programming Task

Implement the gradient of log-softmax in pure Python.

```
import math

def softmax(z):
    m = max(z)
    exps = [math.exp(v - m) for v in z]
    s = sum(exps)
    return [e / s for e in exps]

def E14(z):
    """
    Computes the Jacobian of log-softmax for logits z.
    Returns an n x n matrix.
    """
    p = softmax(z)
    n = len(z)

    J = []
    for i in range(n):
        row = []
        for j in range(n):
            if i == j:
                row.append(1 - p[i])
            else:
                row.append(-p[j])
        J.append(row)

    return J
```