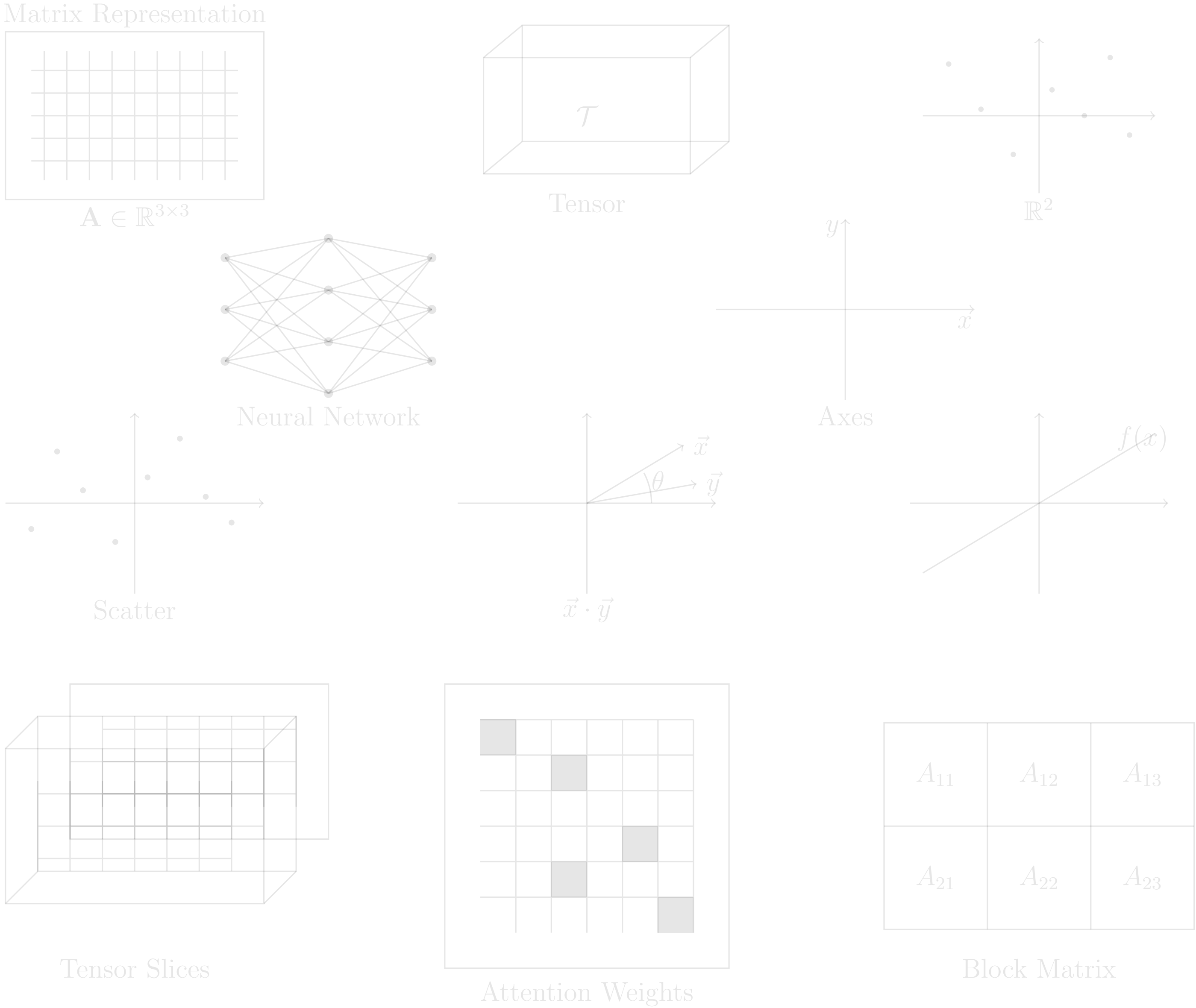
All You Need

Everything You Need to Become an Artificial Intelligence Researcher



Abdulla Ali Khamis Alshamsi

Contents

Introduction of the Volume of Books **1**
- 0.1 Why am I writing this volume of books . . . 1
- 0.2 We must be at the forefront . . . 1
- 0.3 What will this volume of books contain . . . 1

1 Tensor Mathematics and Neural Networks Guide **3**
- 1.1 Introduction . . . 3
  - 1.1.1 What will the book teach you . . . 3
  - 1.1.2 The Chapters (T00 - T08) included in this book . . . 3
- 1.2 T00 - Tensor Algebra Laws . . . 4
  - 1.2.1 Exercise 00 - Tensor Addition Law . . . 4
  - 1.2.2 Exercise 01 - Tensor subtraction Law . . . 5
  - 1.2.3 Exercise 02 - Tensor Hadamard Multiplication Law . . . 6
  - 1.2.4 Exercise 03 - Tensor Division Law . . . 8
  - 1.2.5 Exercise 04 - Tensor Power Law . . . 9
  - 1.2.6 Exercise 05 - Tensor Square Root Law . . . 10
  - 1.2.7 Exercise 06 - Tensor Absolute Value Law . . . 12
  - 1.2.8 Exercise 07 - Tensor Negation Law . . . 13
  - 1.2.9 Exercise 08 - Tensor Sign Law . . . 16
  - 1.2.10 Exercise 09 - Tensor Clamp Law . . . 18
  - 1.2.11 Exercise 10 - Tensor Round Law . . . 20
  - 1.2.12 Exercise 11 - Tensor Floor Law . . . 21
  - 1.2.13 Exercise 12 - Tensor Ceil Law . . . 22

2 Conclusion of the Volume of Books **25**

Conclusion of the Volume of Books **25**
- 2.1 What next . . . 25
- 2.2 Does this volume of books have an end . . . 25

Introduction of the Volume of Books

0.1 Why am I writing this volume of books

One day I decided to learn artificial intelligence programming. What happened was strange; I discovered that the more I delved, the more I realized there was still so much to learn. The deeper I explored this world, the more I understood that I hadn't learned anything, or that what I had learned was just a drop in the ocean. That's why I decided to write down everything I learned on my journey to learning artificial intelligence programming.

What I saw in this science confirmed that we in the UAE need a lot of learning. Therefore, I will use this book to teach future generations in the UAE how to program artificial intelligence, based on the principle of self-learning, just as I implemented it at School 42 Abu Dhabi. What we need is to organize our ideas and begin.

0.2 We must be at the forefront

The competition between Saudi Arabia and the UAE to acquire artificial intelligence is a drop in the ocean compared to what companies like OpenAI, Google, and other major American firms are doing. We must accelerate development because the capabilities these companies possess rival those of most countries in the world. As Emiratis and residents of the UAE, we must expedite the development of this community in programming and advancing advanced artificial intelligence.

It's true we have educational and infrastructure projects, but we lack a huge number of educated personnel. That's why I decided to write this book for them—a solid foundation upon which to build this national workforce, and even for residents of the UAE.

0.3 What will this volume of books contain

Book 1: Tensor Mathematics and Neural Networks Guide

You will learn the fundamentals of mathematics and how to program it using Pytorch. While the first challenge will be easy, the second chapter will be quite difficult if you're new to this field. This isn't because it's impossible to grasp, but because you need four things to overcome the difficulties: first, understand the equation; then, write it using Pytorch; then, ask yourself what its application is in artificial intelligence; and finally, practice creating equations using it. Pay attention: you must complete all four of these points together in every chapter. Don't skip or skip any challenge without completing these four points, because the learning system is built upon what you've learned previously.

Book 1

Tensor Mathematics and Neural Networks Guide

1.1 Introduction

1.1.1 What will the book teach you

Before venturing into the world of AI programming, you must learn the fundamentals of Mathematicals and programming using PyTorch. You will start from the very basics and progress to the most complex mathematical equations. This book will lay the foundation for what's to come, so make sure you understand every lesson and every challenge in this book before moving on to other books.

To emphasize, the book will consist of many exercises, each including a question and a method for finding the solution. The solution will not be provided; you must create it yourself. The book will guide you, as finding the solution is your responsibility. The approach may seem unconventional at times, but this is the foundation of intelligent learning and problem-solving.

1.1.2 The Chapters (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

1.2 T00 - Tensor Algebra Laws

1.2.1 Exercise 00 - Tensor Addition Law

Part 1 - why this matters

Tensor addition is the first operation used to combine numerical information in artificial intelligence systems. It appears in almost every model, from adding bias terms to accumulating gradients during training. Understanding this operation is essential before moving to more complex tensor transformations.

Part 2 - Mathematical form

Mathematical Definition

Let $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{3\times3}$.

Tensor (matrix) addition is a binary operator defined as:

$$+ : \mathbb{R}^{3\times3} \times \mathbb{R}^{3\times3} \to \mathbb{R}^{3\times3}$$

Define:

$$\mathbf{Z} = \mathbf{X} + \mathbf{Y}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times3}$ and the operation is defined component-wise by:

$$Z_{i,j} = X_{i,j} + Y_{i,j} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

Tensor addition in PyTorch is performed using the built-in function `torch.add(x, y)`. This operation applies element-wise addition and requires both tensors to have the same shape.

Before performing any tensor operation, ensure that the inputs are PyTorch tensors created using `torch.tensor()`. Understanding how tensors are constructed is essential before manipulating them.

PyTorch tensors can exist on different devices:

- CPU (default)
- GPU (CUDA-enabled devices)

When working with multiple tensors, both tensors must be located on the same device. Mixing CPU and GPU tensors will result in a runtime error.

To deepen your understanding, explore:

- How to create tensors using `torch.tensor()`
- How to check the device of a tensor
- How to move tensors between CPU and GPU

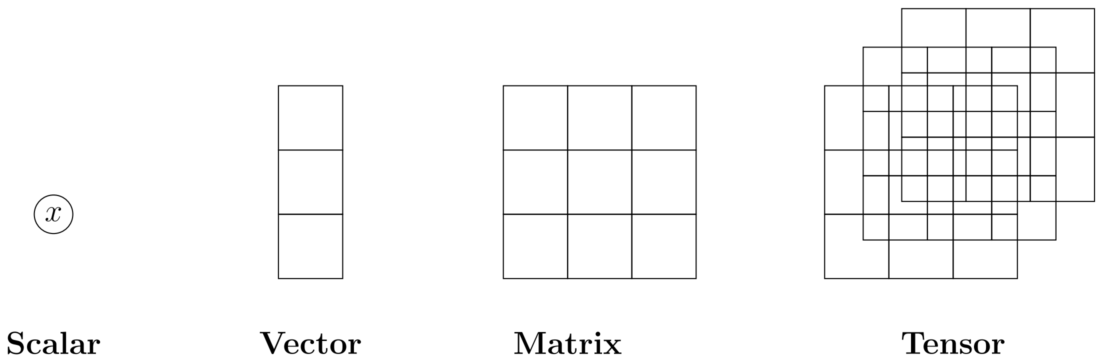
This exercise assumes CPU-based tensors, but the same logic applies to GPU tensors when explicitly moved to a CUDA device.

Part 4 - Python code skeleton

```
import torch

def E00(x,y):
    z = torch...
    return z

def main():
    x = torch.tensor...
    y = torch.tensor...
    print(E00(...))

if __name__ == "__main__":
    main()
```

1.2.2 Exercise 01 - Tensor subtraction Law

Part 0 - Before we begin



All numerical data in artificial intelligence is ultimately represented using the same underlying structure; the difference lies only in how many dimensions the data has.

- **Scalar**: A single numerical value with no dimensions.
- **Vector**: A one-dimensional collection of ordered values.
- **Matrix**: A two-dimensional grid of values arranged in rows and columns.
- **Tensor**: A generalization to three or more dimensions, used to represent complex data such as images, sequences, or batches.

Part 1 - why this matters

This operation has the same importance as the previous exercise. The only difference is that subtraction is used instead of addition. Understanding this change is essential before moving to more advanced tensor operations.

Part 2 - Mathematical form

Let $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{3\times 3}$.

Tensor (matrix) subtraction is a binary operator defined as:

$$- : \mathbb{R}^{3\times 3} \times \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \mathbf{X} - \mathbf{Y}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the operation is defined component-wise by:

$$Z_{i,j} = X_{i,j} - Y_{i,j} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, tensor subtraction is performed using the built-in function `torch.sub(x, y)`. The reader is encouraged to search for this function and understand how it applies element-wise subtraction between tensors.

Part 4 - Python code skeleton

```
import torch

def E01(x,y):
    z = torch...
    return z

def main():
    x = torch.tensor...
    y = torch.tensor...
    print(E01(...))

if __name__ == "__main__":
    main()
```

1.2.3 Exercise 02 - Tensor Hadamard Multiplication Law

Part 1 - why this matters

This exercise continues the logical progression of tensor operations. After learning how tensors are combined through addition and how differences are expressed through subtraction, Hadamard multiplication introduces a new concept: controlling how information is scaled rather than simply accumulated or removed.

Unlike addition and subtraction, element-wise multiplication directly modifies the strength of each individual value. Each component can be amplified, reduced, or completely suppressed depending on the corresponding value in the second tensor. This changes the behavior of the data itself, not just its numerical total.

Understanding this operation is essential because it represents a foundational mechanism used later in more advanced tensor transformations. Mastering this concept at an early stage makes the transition to complex model structures more intuitive and coherent.

Part 2 - Mathematical form

Let $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{3\times 3}$.
Hadamard multiplication (element-wise multiplication) is a binary operator defined as:

$$\odot : \mathbb{R}^{3\times 3} \times \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \mathbf{X} \odot \mathbf{Y}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the operation is defined component-wise by:

$$Z_{i,j} = X_{i,j} \cdot Y_{i,j} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, Hadamard (element-wise) multiplication can be performed using the built-in operator `x * y`.

The reader is encouraged to search for a function called `torch.mul()` and understand how it performs element-wise multiplication between tensors.

Part 4 - Python code skeleton

```
import torch

def E02(x,y):
    z = torch...
    return z

def main():
    x = torch.tensor...
    y = torch.tensor...
    print(E02(...))

if __name__ == "__main__":
    main()
```

1.2.4 Exercise 03 - Tensor Division Law

Part 0 - The weight

In artificial intelligence, a weight represents the strength of influence that an input has on the model's internal computation and final decision. Each input signal is multiplied by a weight, allowing the model to emphasize important information and suppress irrelevant or noisy signals. Learning in neural networks is therefore the process of continuously adjusting these weights to reflect how much the model should trust each input.

$$2000 \times 2000 = 4{,}000{,}000$$

$$0.2 \times 0.2 = 0.04$$

The large numerical gap produced by unscaled weights leads to unstable matrix computations, where values grow too fast and dominate subsequent operations. Keeping weights within a controlled and close numerical range ensures stable tensor operations, preserves relative relationships between features, and allows learning algorithms to update weights smoothly without numerical explosion or loss of precision.

Part 1 - why this matters

Tensor Division Law is required because repeated multiplication and accumulation in neural networks cause values to either explode or vanish, and division is used to rescale tensors into a numerical range where learning remains stable.

In normalization, division is not applied merely to modify data values, but to decouple magnitude from structure, allowing the model to learn patterns rather than be dominated by raw numerical scale.

When working with ratios, probabilities, and feature weights, division transforms absolute values into relative relationships, enabling meaningful comparison and scale-independent decision making.

Part 2 - Mathematical form

Let $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{3\times3}$.
Tensor Division Law (element-wise):

$$\oslash : \mathbb{R}^{3\times3} \times \mathbb{R}^{3\times3} \to \mathbb{R}^{3\times3}$$

Define:

$$\mathbf{Z} = \mathbf{X} \oslash \mathbf{Y}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times3}$ and the law is defined component-wise by:

$$Z_{i,j} = \frac{X_{i,j}}{Y_{i,j}} \quad \text{for } i, j \in \{1, 2, 3\}, \quad Y_{i,j} \neq 0.$$

Part 3 - Hints and Helper

In PyTorch, tensor division (element-wise division) can be performed using the built-in operator `x / y`.

The reader is encouraged to search for a function called `torch.div()` and understand how it performs element-wise division between tensors.

Part 4 - Python code skeleton

```
import torch

def E03(x,y):
    z = torch...
    return z

def main():
    x = torch.tensor...
    y = torch.tensor...
    print(E03(...))

if __name__ == "__main__":
    main()
```

1.2.5 Exercise 04 - Tensor Power Law

Part 0 - The Magnitude

In artificial intelligence, knowing whether a value is positive or negative is often less important than understanding its magnitude relative to zero. Magnitude represents the size or intensity of an error or deviation without regard to direction, since a model must measure how large an error is rather than where it lies.

For this reason, power-based operations are used to represent magnitude in a numerically stable form, enabling reliable measurement and optimization of error and deviation.

Part 1 - why this matters

In artificial intelligence, it is essential to distinguish between the influence of a signal controlled by weights and the magnitude of a value measured independently of its direction.

The Tensor Power Law is used to convert values into measurable magnitudes, enabling stable numerical evaluation of error and deviation.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Power Law (element-wise) is defined as:

$$\circledast : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \mathbf{X} \circledast 2$$

where $\mathbf{Z} \in \mathbb{R}^{3\times3}$ and the law is defined component-wise by:

$$Z_{i,j} = (X_{i,j})^2 \quad \text{for } i,j \in \{1,2,3\}.$$

Part 3 - Hints and Helper

In PyTorch, tensor power (element-wise exponentiation) can be performed using the built-in operator `x ** 2`.

The reader is encouraged to search for a function called `torch.pow()` and understand how it applies element-wise power operations between tensors or between a tensor and a scalar.

Part 4 - Python code skeleton

```
import torch

def E04(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E04(...))

if __name__ == "__main__":
    main()
```

1.2.6 Exercise 05 - Tensor Square Root Law

Part 0 - The element-wise

What "element-wise" means. An **element-wise** operation applies the same rule to each entry of a tensor *independently*, so the value at one position is computed only from the value(s) at the *same position*. As a result, the output typically has the **same shape** as the input (or the broadcasted shape if broadcasting is used).

Shape requirement (matrix × matrix). For two matrices, an element-wise binary law is defined only when the shapes match:

$$\odot : \mathbb{R}^{3\times3} \times \mathbb{R}^{3\times3} \to \mathbb{R}^{3\times3}, \qquad Z_{i,j} = X_{i,j} \odot Y_{i,j}.$$

Here, each output entry $Z_{i,j}$ depends only on $X_{i,j}$ and $Y_{i,j}$.

Why we use element-wise laws in AI. Element-wise laws are used to transform or scale data *without mixing information* across positions. They are essential for operations such as normalization, magnitude measurement (e.g., squaring), and pointwise non-linear transformations, where we want to adjust values while preserving the tensor structure.

Why $\mathbb{R}^{3\times3}$ and $\mathbb{R}^{3\times2}$ is not element-wise. If $\mathbf{X} \in \mathbb{R}^{3\times3}$ and $\mathbf{Y} \in \mathbb{R}^{3\times2}$, then an element-wise binary operation is **not defined** because there is no one-to-one alignment between entries:

$$\mathbb{R}^{3\times3} \;\not\odot\; \mathbb{R}^{3\times2} \quad \text{(shape mismatch)}.$$

So your intuition is correct: this is **not** an element-wise operation. However, this does **not** automatically mean it is a different "type of operation" by itself; it is simply a **shape mismatch** for element-wise binary laws. (Only if a broadcasting rule applies can some mismatched shapes still work, but 3×3 and 3×2 do not broadcast to a common shape.)

Part 1 - why this matters

In artificial intelligence, operations like squaring are used to measure intensity or error without being affected by sign, but they also change the measurement unit and push values into a much larger numerical range than the original scale. The Tensor Square Root Law restores values back to their natural scale after magnitude-based computations, making results interpretable and comparable by returning them to the same "unit" as the original signal rather than leaving them in squared units. Additionally, the square root compresses the numerical range and mitigates the growth caused by squaring and accumulation, improving tensor-level numerical stability and reducing the risk of explosion or precision loss during training.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$ with $X_{i,j} \geq 0$.

Tensor Square Root Law (element-wise) is defined as:

$$\sqrt{\cdot} : \mathbb{R}^{3\times 3}_{\geq 0} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \sqrt{\mathbf{X}}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = \sqrt{X_{i,j}} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, tensor square root (element-wise square root) can be performed using the built-in function `torch.sqrt()`.

The reader is encouraged to explore how `torch.sqrt()` applies the square root operation element-wise on tensors and to understand its role in restoring scale after magnitude-based operations.

Part 4 - Python code skeleton

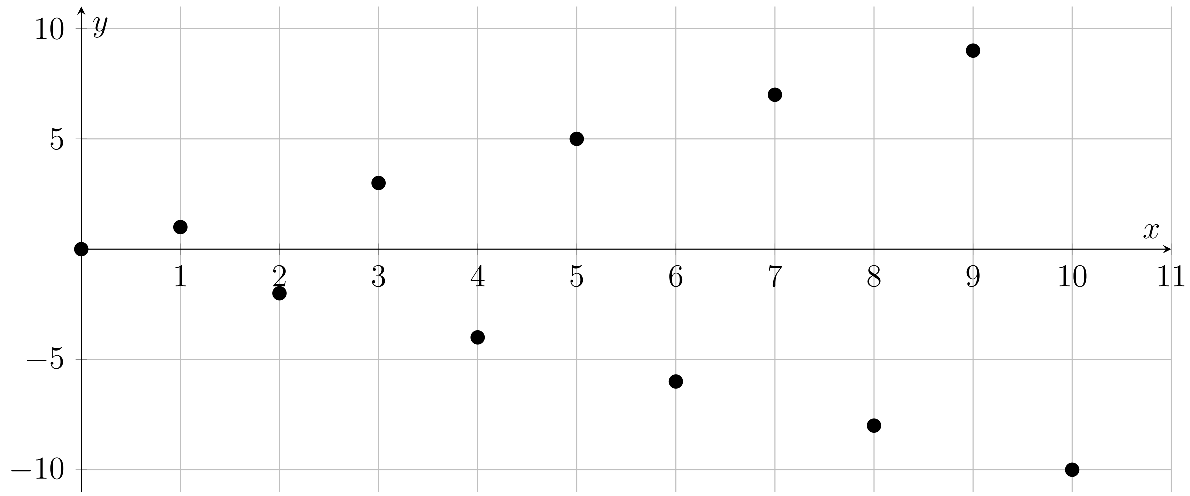
```
import torch

def E05(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E05(...))

if __name__ == "__main__":
    main()
```

1.2.7 Exercise 06 - Tensor Absolute Value Law

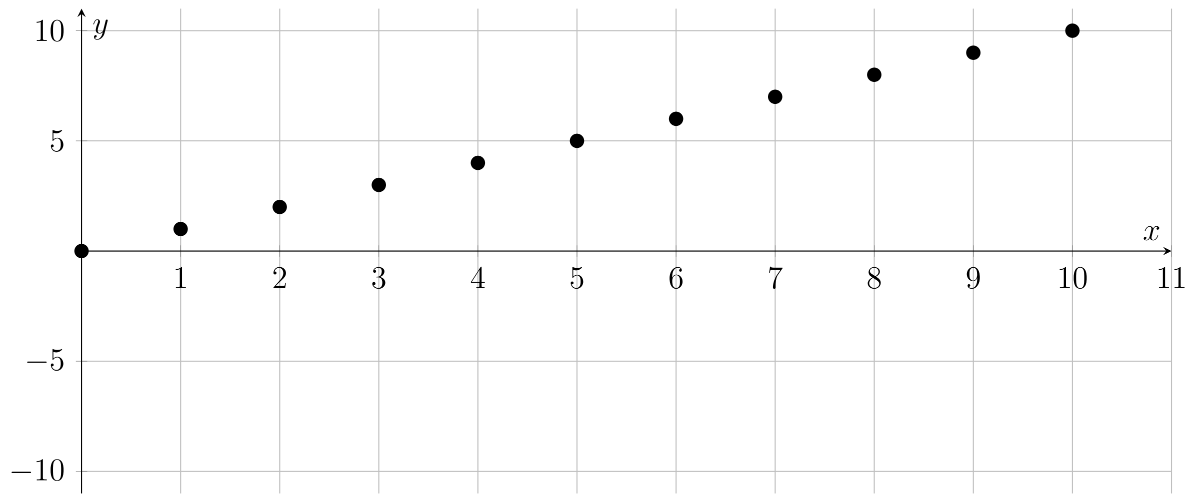
Part 1 - why this matters

In artificial intelligence equations, the goal is not always to determine the direction of an error or the sign of a value, but rather to measure its true magnitude regardless of whether it is positive or negative. Positive and negative values represent different directions around a reference point, but when evaluating model performance or computing deviation, the sign can be misleading because it allows errors to cancel each other out during aggregation. For this reason, it becomes necessary to remove the sign and convert values to non-negative quantities, so that every deviation is counted as an independent contribution reflecting its actual intensity rather than its direction. This transformation ensures that no error disappears simply because an opposite error exists.

Points without Absolute Value



Points with Absolute Value



The absolute value provides a precise and straightforward way to achieve this goal without altering the numerical scale of values or artificially amplifying them. Unlike squaring, which changes the measurement unit and exaggerates large differences, the absolute value preserves linear relationships between values and enables direct, numerically stable measurement of deviation. This is why it is widely used in artificial intelligence equations whenever a fair and balanced assessment of error or distance is required, especially in scenarios where excessive influence from large values must be avoided or clear interpretability of results is desired. In this sense, removing the sign is not a mathematical simplification, but a deliberate choice to separate measurement from direction within the model.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Absolute Value Law (element-wise) is defined as:

$$|\cdot| : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = |\mathbf{X}|$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = |X_{i,j}| \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor absolute value (element-wise absolute value) can be computed using the built-in function `torch.abs()`.

The reader is encouraged to explore how `torch.abs()` removes the sign of each element independently and how it is commonly used in error measurement and magnitude-related computations.

Part 4 - Python code skeleton

```
import torch

def E06(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E06(...))

if __name__ == "__main__":
    main()
```

1.2.8 Exercise 07 - Tensor Negation Law

Part 0.1 - Optimization and Gradients

In artificial intelligence, the term Optimization is not used in its general linguistic sense. Instead, it refers to the controlled adjustment of a model's numerical parameters within a high-dimensional mathematical space. Researchers treat the model as a purely numerical system governed by a single objective function, and optimization defines how this system evolves over time toward configurations that reduce error according to a predefined criterion.

From a research perspective, Optimization transforms learning into a process that can be precisely managed and analyzed. Rather than being an abstract or emergent behavior, learning becomes a sequence of small, measurable parameter updates. This framing allows researchers to design learning algorithms whose behavior can be studied, compared, and refined, making model training both predictable and interpretable.

Within artificial intelligence research, the Gradient is understood as a mathematical sensing mechanism rather than merely a derivative. It provides local information about how sensitive the model is to small changes in its internal parameters, revealing the directions in which the objective function changes most rapidly.

In practice, the Gradient serves as raw directional information rather than a decision rule. Researchers do not follow it blindly; instead, they manipulate its influence through additional operations such as negation, scaling, or constraint enforcement. Thus, the Gradient supplies the signal, while Optimization defines how that signal is used to guide the learning process.

Part 0.2 - Difference and Error Function

In artificial intelligence, Difference is not understood as a simple subtraction operation, but as a foundational concept that signals the existence of a gap between two states: what the model produces and what it is expected to produce. Researchers treat Difference as the first mathematical acknowledgment that the current system deviates from its objective, without yet explaining the cause of that deviation.

From a research standpoint, Difference is the most primitive representation of error before any further processing. Once the difference is formulated, it can be treated as an independent mathematical object that can be scaled, transformed, sign-inverted, or passed through additional functions. For this reason, Difference is not the end of computation but its starting point, forming the basis for all subsequent correction steps.

The Error Function represents the next conceptual layer beyond Difference. In the researcher's view, an Error Function is a deliberate definition of what constitutes error and how severely it should be penalized. It is not merely a computational formula, but a formal policy that encodes the learning objective and determines which deviations matter and which can be ignored.

At a deeper level, the Error Function governs the long-term behavior of the model. It shapes the optimization landscape by deciding how raw differences are aggregated and weighted, thereby influencing the direction and stability of learning. As a result, changing the Error Function can fundamentally alter the learning dynamics of a system, even when the data and model architecture remain unchanged.

Part 0.3 - Signal directions and Opposition

In artificial intelligence, the direction of a signal is as important as its numerical magnitude. Researchers treat each signal as a force that pushes the model in a specific direction within the solution space. The same numerical value can be beneficial or harmful depending on its sign, which means that model behavior cannot be understood by magnitude alone without considering direction.

From a training perspective, signal direction determines whether a proposed change moves the model closer to its objective or farther away from it. For this reason, many learning procedures prioritize controlling the direction of change before adjusting its strength. An incorrect direction, even if small, can accumulate over time and lead to significant divergence during training.

The concept of Opposition in artificial intelligence does not imply rejection, but rather a deliberate decision to act against a given signal direction. Researchers recognize that certain signals may carry valid information about the system while still leading to undesirable behavior if followed directly. Opposition allows this information to be used while preventing harmful movement.

At a deeper research level, opposition represents the point at which control is imposed on model behavior. Through operations such as negation, the learning process can exploit directional information without being dominated by it. This idea underlies practices like reversing the gradient during optimization and explains why tensor-level operations such as negation appear early and repeatedly in learning algorithms.

Part 1 - why this matters

The primary purpose of the Tensor Negation Law is to provide a simple and consistent way to reverse the effect of values without altering the structure or layout of the data. In many artificial intelligence systems, we do not need to introduce new information; instead, we need to express that an existing contribution should act in the opposite direction. Negation achieves this by preserving the same tensor shape and positions while inverting the sign of each value.

Tensor negation is used extensively because many core operations are based on comparison and correction. When computing differences between model outputs and target values, or when updating parameters during training, negation is required to move against an undesired direction. Without negation, it would be impossible to formally represent error, opposition, or corrective movement in a clean and structured mathematical way.

Most importantly, Tensor Negation preserves the tensor's internal structure. The dimensionality, alignment, and positional meaning of the data remain unchanged, while only the sign is inverted. This makes negation a safe and essential operation in learning systems, as it modifies numerical behavior without breaking the integrity of tensor-based representations or mixing information across locations.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Negation Law (element-wise) is defined as:

$$- : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = -\mathbf{X}$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = -\,X_{i,j} \quad \text{for } i,j \in \{1,2,3\}.$$

Part 3 - Hints and Helper

In PyTorch, tensor negation (element-wise sign inversion) can be performed using the built-in function `torch.neg()`, or equivalently by applying the unary minus operator `-x`.

The reader is encouraged to explore how unary negation acts independently on each tensor entry without changing the tensor shape, and to compare the explicit use of `torch.neg()` with the operator form to understand their equivalence at the tensor level.

Part 4 - Python code skeleton

```
import torch

def E07(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E07(...))

if __name__ == "__main__":
    main()
```

1.2.9 Exercise 08 - Tensor Sign Law

Part 1 - why this matters

In artificial intelligence systems, numerical values are constantly used to control decisions, updates, and internal signals. However, not every operation requires knowing the exact magnitude of a value. In many situations, what truly matters is the *direction* of influence rather than its size. Positive and negative values indicate opposite effects within the model, and identifying this direction alone is often sufficient to guide learning or adjustment processes.

Neural networks, in particular, repeatedly update their parameters based on signals that may vary widely in scale. Large values can cause unstable behavior, while very small values may become negligible during aggregation. By extracting only the sign of a tensor, the model can ignore harmful scale differences and focus solely on whether each component pushes the system forward, backward, or not at all. This leads to more stable and interpretable behavior in parts of the learning process where controlled directional movement is preferred over exact measurement.

Mathematically, the Tensor Sign Law allows the model to separate *decision* from *measurement*. Instead of treating numbers as precise quantities, the law converts them into directional indicators. This is especially useful in optimization and control-related operations, where consistent movement in the correct direction is more important than reacting strongly to large but possibly noisy values.

The Tensor Sign Law can be expressed in its simplest form as:

$$z = \text{sign}(x)$$

where:

- x represents a real-valued scalar or a single element of a tensor.
- $\text{sign}(x)$ extracts the direction of x.
- $z = 1$ if $x > 0$, indicating a positive direction.
- $z = -1$ if $x < 0$, indicating a negative direction.
- $z = 0$ if $x = 0$, indicating no directional influence.

When applied element-wise to tensors, this operation enables neural networks to reason about directional behavior independently of magnitude, forming a foundation for stable updates, directional analysis, and simplified control mechanisms within artificial intelligence models.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Sign Law (element-wise) is defined as:

$$\text{sign}(\cdot) : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \text{sign}(\mathbf{X})$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = \begin{cases} 1 & \text{if } X_{i,j} > 0 \\ 0 & \text{if } X_{i,j} = 0 \\ -1 & \text{if } X_{i,j} < 0 \end{cases} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor sign operation (element-wise sign extraction) can be computed using the built-in function `torch.sign()`.

The reader is encouraged to explore how `torch.sign()` separates *direction* from *magnitude*, returning only the sign of each element, and how this operation is commonly paired with absolute value or norm based laws in optimization, gradient analysis, and control of update direction.

Part 4 - Python code skeleton

```
import torch

def E08(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E08(...))

if __name__ == "__main__":
    main()
```

1.2.10 Exercise 09 - Tensor Clamp Law

Part 0 - conceptual overview

In a neural network, all numerical behavior revolves around a small set of interconnected quantities: weights, activations, scores, and gradients. These elements do not exist independently; instead, they form a continuous feedback loop that drives learning and decision-making within the model.

Weights represent the internal parameters of the network and determine how input information is transformed. Activations are the intermediate signals produced when inputs are combined with weights and passed through nonlinear functions. Scores are the final numerical outputs of the network before interpretation, often representing confidence, preference, or likelihood. Together, weights and activations shape the scores produced by the model.

Gradients complete the loop by measuring how changes in weights affect the final scores and error. They flow backward through the network and guide how weights should be adjusted. Because activations, scores, and gradients can grow unbounded or become unstable, controlling their numerical range becomes essential. Understanding this relationship explains why mechanisms such as clamping are needed to keep all components operating within safe and meaningful limits.

Part 1 - why this matters

Artificial intelligence systems continuously generate numerical values while processing data, updating parameters, and producing outputs. These values are not inherently constrained and can easily grow too large or become too small during training or inference. When such uncontrolled values propagate through a neural network, they may cause numerical instability, saturation, or unreliable behavior that degrades the overall performance of the model.

The Tensor Clamp Law introduces explicit boundaries that restrict values to a predefined and meaningful range. By enforcing lower and upper limits, the model is protected from extreme values that do not carry additional useful information. This controlled restriction allows learning processes to remain stable while still preserving the essential structure of the data. Instead of eliminating information, clamping preserves valid values and only corrects those that exceed acceptable limits.

Within neural networks, clamping is commonly used to maintain safe operating ranges for activations, gradients, and outputs. It ensures that numerical values remain interpretable and physically or logically valid throughout computation. In this sense, the Tensor Clamp Law acts as a stabilizing mechanism that enables artificial intelligence systems to learn effectively without being disrupted by unbounded numerical behavior.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$ and let $a, b \in \mathbb{R}$ with $a \leq b$.

Tensor Clamp Law (element-wise) is defined as:

$$\mathrm{clamp}_{[a,b]}(\cdot) : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \mathrm{clamp}_{[a,b]}(\mathbf{X})$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = \begin{cases} a & \text{if } X_{i,j} < a \\ X_{i,j} & \text{if } a \leq X_{i,j} \leq b \\ b & \text{if } X_{i,j} > b \end{cases} \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor clamp operation (element-wise value limiting) can be computed using the built-in function `torch.clamp()`.

The reader is encouraged to explore how `torch.clamp()` enforces lower and upper bounds on each tensor element independently, preventing values from exceeding predefined limits and ensuring numerical stability during computation.

Part 4 - Python code skeleton

```
import torch

def E09(x, m, M):
    z = torch...
    return z

def main():
    x = torch.tensor...
    m =...
    M =...
    print(E09(...))

if __name__ == "__main__":
    main()
```

1.2.11 Exercise 10 - Tensor Round Law

Part 1 - why this matters

Neural networks operate primarily in a continuous numerical space, where values such as weights, activations, and scores can take any real number. This continuous representation is essential for learning and optimization, but it does not always align with the requirements of decision-making or real-world execution. In many situations, a model must eventually produce discrete, integer-based outcomes rather than continuous values.

The Tensor Round Law provides a controlled way to convert continuous values into stable integer representations. By rounding values to the nearest integer, small numerical fluctuations are removed without introducing large distortions. This is particularly important when values are close to decision boundaries, where minor changes could otherwise cause inconsistent or unstable behavior.

Within artificial intelligence systems, rounding is commonly used when transitioning from internal numerical processing to final outputs, counters, or index-based operations. It enables models to move from soft, approximate representations to clear and interpretable decisions, ensuring that continuous learning processes can be safely integrated with discrete system requirements.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times3}$.
Tensor Round Law (element-wise) is defined as:

$$\mathrm{round}(\cdot) : \mathbb{R}^{3\times3} \to \mathbb{R}^{3\times3}$$

Define:

$$\mathbf{Z} = \mathrm{round}(\mathbf{X})$$

where $\mathbf{Z} \in \mathbb{R}^{3\times3}$ and the law is defined component-wise by:

$$Z_{i,j} = \mathrm{round}(X_{i,j}) \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor rounding operation (element-wise rounding to the nearest integer) can be computed using the built-in function `torch.round()`.

The reader is encouraged to explore how `torch.round()` converts continuous values into discrete ones by rounding each element independently, and how this operation is commonly used when transitioning from continuous representations to discrete decisions or stabilized numerical outputs.

Part 4 - Python code skeleton

```
import torch

def E10(x):
```

```
        z = torch...
        return z

    def main():
        x = torch.tensor...
        print(E10(...))

    if __name__ == "__main__":
        main()
```

1.2.12 Exercise 11 - Tensor Floor Law

Part 1 - why this matters

Artificial intelligence systems often work with continuous numerical values, yet many decisions and operations within these systems require discrete and strictly bounded outcomes. In such cases, fractional values cannot be interpreted directly, and a clear rule is needed to convert continuous quantities into valid integers without violating system constraints.

The Tensor Floor Law provides a conservative transformation by mapping each value to the largest integer that is less than or equal to it. This guarantees that the result never exceeds the original value, making it suitable for situations where upper limits must be respected. By consistently rounding downward, the operation prevents unintended overestimation that could lead to invalid decisions or resource misuse.

Within artificial intelligence pipelines, floor-based transformations are used when values represent counts, indices, or execution steps that must remain within safe and allowable bounds. The Tensor Floor Law ensures that continuous numerical outputs can be safely translated into discrete, actionable values while preserving logical consistency throughout the system.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Floor Law (element-wise) is defined as:

$$\lfloor \cdot \rfloor : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \lfloor \mathbf{X} \rfloor$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = \lfloor X_{i,j} \rfloor \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor floor operation (element-wise rounding down to the nearest integer) can be computed using the built-in function `torch.floor()`.

The reader is encouraged to explore how `torch.floor()` consistently rounds values downward regardless of their fractional part, and how this operation is commonly used when enforcing conservative or lower-bound numerical decisions in artificial intelligence systems.

Part 4 - Python code skeleton

```
import torch

def E11(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E11(...))

if __name__ == "__main__":
    main()
```

1.2.13 Exercise 12 - Tensor Ceil Law

Part 0 - Conservation and Safeguarding

Artificial intelligence systems are fundamentally numerical systems that rely on continuous values to represent uncertainty, estimates, and learned behavior. However, these systems do not operate in isolation; they must eventually interact with finite computational resources, discrete execution steps, and real-world constraints. As a result, numerical decisions in artificial intelligence are not only mathematical choices but also design decisions that reflect how the system handles risk, error, and failure.

Two fundamental design philosophies emerge when continuous values must be converted into discrete decisions: *conservation* and *safeguarding*. These philosophies define how an artificial intelligence system reacts when faced with uncertainty in numerical estimation. Rather than optimizing for mathematical elegance, they optimize for system reliability and practical correctness.

Conservation represents a cautious approach that prioritizes efficiency and resource control. Under this philosophy, the system avoids allocating or executing more than what is strictly justified by the current numerical estimate. The underlying assumption is that overestimation is more harmful than underestimation. Conservation is therefore used when excessive allocation, unnecessary computation, or inflated execution can degrade performance, increase cost, or destabilize the system.

Safeguarding represents a complementary but fundamentally different philosophy. Instead of minimizing usage, it prioritizes completeness and correctness. Under this approach, the system ensures that requirements are fully met, even if this means allocating slightly more resources or executing additional steps. The core assumption of safeguarding is that underestimation is more dangerous than overestimation, especially when missing data, incomplete processing, or early termination can lead to silent failures.

In artificial intelligence development, neither philosophy is universally superior. Each reflects a different tolerance for risk. Conservation accepts the possibility of doing less in exchange for efficiency, while safeguarding accepts controlled excess in exchange for reliability. The choice between them is not a mathematical preference but a system-level judgment about which type of failure is unacceptable in a given context.

These philosophies directly influence how numerical transformations are applied within neural networks and AI pipelines. When values represent optional capacity, iterative refinement, or expandable processes, conservation-oriented decisions are often appropriate. When values represent mandatory coverage, irreversible decisions, or completeness requirements, safeguarding-oriented decisions become essential.

Understanding conservation and safeguarding as design principles allows AI developers to reason more clearly about numerical transformations such as floor and ceiling operations. Rather than viewing these operations as simple rounding rules, they can be understood as explicit expressions of system intent, encoding how the model chooses to balance efficiency against reliability within its computational logic.

Part 1 - why this matters

Artificial intelligence systems frequently work with continuous numerical values that represent estimated quantities such as required steps, resource usage, or data coverage. However, many execution-level decisions within these systems cannot operate on fractional values and must rely on integer-based quantities that fully satisfy the underlying requirement.

The Tensor Ceil Law provides a safe transformation by mapping each value to the smallest integer greater than or equal to it. This guarantees that the resulting value never underestimates the original quantity. By enforcing upward rounding, the model avoids incomplete processing, missed data, or insufficient allocation that could otherwise occur when fractional estimates are truncated.

Within artificial intelligence pipelines, ceiling-based transformations are used when full coverage and completeness are more important than minimizing numerical overhead. The Tensor Ceil Law ensures that continuous estimates are converted into discrete decisions that meet or exceed required thresholds, enabling reliable and robust system behavior.

Part 2 - Mathematical form

Let $\mathbf{X} \in \mathbb{R}^{3\times 3}$.

Tensor Ceil Law (element-wise) is defined as:

$$\lceil\cdot\rceil : \mathbb{R}^{3\times 3} \to \mathbb{R}^{3\times 3}$$

Define:

$$\mathbf{Z} = \lceil \mathbf{X} \rceil$$

where $\mathbf{Z} \in \mathbb{R}^{3\times 3}$ and the law is defined component-wise by:

$$Z_{i,j} = \lceil X_{i,j} \rceil \quad \text{for } i, j \in \{1, 2, 3\}.$$

Part 3 - Hints and Helper

In PyTorch, the tensor ceiling operation (element-wise rounding up to the nearest integer) can be computed using the built-in function `torch.ceil()`.

The reader is encouraged to explore how `torch.ceil()` consistently maps each value to the smallest integer greater than or equal to it, and how this operation is used when enforcing upward-safe or minimum-satisfying numerical decisions in artificial intelligence systems.

Part 4 - Python code skeleton

```
import torch

def E12(x):
    z = torch...
    return z

def main():
    x = torch.tensor...
    print(E12(...))

if __name__ == "__main__":
    main()
```

2

Conclusion of the Volume of Books

2.1 What next

The real goal is to delve deeper into artificial intelligence, machine learning, deep learning, and perhaps even self-programming in the future. Explore the philosophy of building mathematical models and move away from the notion that artificial intelligence will become uncontrollable. Research seeks solutions and development, not media attention. Focus on your research in artificial intelligence and create value in this world. What you've learned in these books is just a drop in the ocean. Artificial intelligence is like a black hole; once you get close to it, you'll be fascinated. And the strange thing is, you won't know if it has an end. Build your future with your own hands. Place yourself among the researchers. You are responsible for developing artificial intelligence. Be a role model for others.

2.2 Does this volume of books have an end

Yes, it has an end, and that's when the author dies. What I do is compile what I've learned throughout my life. Emirati society is my responsibility; I try to build a bright future for it, placing it among the ranks of the developed world. I believe we must expand our knowledge and compete with the rest of the world. The future must be bright, and bright, for the modern United Arab Emirates.