MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

CENG 403

Spring 2024

Take-Home Exam 1

REGULATIONS

Due date: 23:59, 10 March, 2024, Sunday *(Not subject to postpone)*

Submission: Electronically. You should save your program source code as a text file named `cergen.py`. Check the announcement on the ODTUCLASS course page for the submission procedure.

Team: There is **no** teaming up. This is an EXAM.

Cheating: Source(s) and Receiver(s) will receive zero and be subject to disciplinary action. You are NOT allowed to use code pieces from the Internet or AI tools.

1 OBJECTIVE

The main objective of this take home exam (THE) is to implement your own tensor library, called CerGen (short for CENG Gergen – gergen: one of the Turkish translations of the term tensor). The aim of this THE is to make sure that you are comfortable with the mathematical foundations required for deep learning. Moreover, this THE will provide you with practical coding experience without relying on external libraries like NumPy or PyTorch for a solid foundation.

2 TASK DESCRIPTION

In this THE, you will develop the `gergen` class, a custom data structure designed to provide a hands-on experience with fundamental array, matrix and tensor operations, mirroring some functionalities typically found in libraries like NumPy and PyTorch. The `gergen` class encapsulates a dynamic array, allowing for operations such as element-wise mathematical functions, matrix transformations, and basic linear algebra computations.

2.1 Fundamental Operations

Before diving into the implementation of the `gergen` class within the `CerGen` tensor library, you will implement some fundamental functionalities that will support the operations within the `gergen` class.

- `cekirdek(sayi: int) -> None`: Sets the seed for random number generation to ensure reproducibility of results. Before generating random numbers (for instance, when initializing tensors with random values), you can call this function to set the seed.
- `rastgele_dogal(boyut, aralik=(0, 100), dagilim='uniform') -> gergen`: Generates a `gergen` of specified dimensions with random integer values. The `boyut` parameter is a tuple specifying the dimensions of the `gergen` to be generated. The `aralik` parameter is an

optional tuple (min, max) specifying the range of random values, with a default range of (0, 100). The `dagilim` parameter specifies the distribution of random values, with 'uniform' as the default distribution. Possible values for `dagilim` include 'uniform' for a uniform distribution. You should raise ValueError if `dagilim` parameter is given differently.

- `rastgele_gercek(boyut, aralik=(0.0,1.0), dagilim=None) -> gergen`: Generates a `gergen` of specified dimensions with random floating-point values. The `boyut` parameter is a tuple specifying the dimensions of the `gergen` to be generated. The `aralik` parameter is an optional tuple (min, max) specifying the range of random values, with a default range of (0.0,1.0). The `dagilim` parameter specifies the distribution of random values, with 'uniform' as the default distribution. Possible values for `dagilim` include 'uniform' for a uniform distribution. You should raise ValueError if `dagilim` parameter is given differently.

2.2 The `gergen` Class

The `gergen` class is a custom implementation of a tensor, similar to the PyTorch Tensor. Below are the key features and member functions of the `gergen` class, each designed to introduce different aspects of array/tensor handling operations:

- The class is initialized with `veri`, which can be either a scalar (int/float) for simple numerical values or a list for representing vectors, matrices, or higher-dimensional tensors, enabling versatile handling of both simple and complex numerical data structures.
- The **attributes** of the class include:
  1. `veri`: Can be either a scalar (int/float) or a nested list of numbers, serving as the foundational data for the `gergen` object.
  2. `D`: Stores the transpose of `gergen`. D is short for devrik (transpose in Turkish).
  3. `boyut`: A tuple that records the dimensions of the `gergen`, representing its shape in terms of rows, columns, and potentially further dimensions for higher-order tensors.
- **Key Methods** include:
  1. `__init__(self, veri=None) -> None`: Constructor for the `gergen` class, initializing a new `gergen` instance with the data provided in `veri`, which is optional and defaults to `None` if not specified.
  2. `__getitem__(self, index) -> gergen`: Enables indexing the `gergen`, allowing access to specific elements, rows, columns, or sub-tensors using standard Python indexing syntax. This method will be called when an index is used on an instance of the `gergen` class, such as `g[0]` for a `gergen` `g`.
  3. `__str__(self) -> str`: Generates a string representation of the `gergen`. When `print(g1)` is invoked on a `gergen` instance `g1`, this method is automatically called, resulting in the display of the tensor's `boyut` followed by its `veri`. For example, if `g1` is generated with list `[[1,2,3],[4,5,6]]`, `print(g1)` should return:

```
2x3 boyutlu gergen:
[[1, 2, 3]
[4, 5, 6]]
```

4. `__mul__(self, other: Union[gergen, int, float])` → `gergen`: This method facilitates the multiplication of the `gergen` either with another `gergen` instance for element-wise multiplication, or with a scalar (int/float), yielding a new `gergen` object as the result. The `other` parameter is permitted to be a `gergen`, an integer, or a floating-point number. Error handling is incorporated to manage cases where the `other` parameter is neither a `gergen` object nor a numerical scalar. If the dimensions of two `gergen` instances do not align for element-wise multiplication, or if an incompatible type is provided for `other`, a `TypeError` or `ValueError` is raised.
5. `__truediv__(self, other: Union[gergen,int, float])` → `gergen`: This method implements division for the `gergen`, facilitating element-wise division by a scalar (an integer or a float), and encapsulates the result in a new `gergen` instance. True division is employed, ensuring that the result is always a floating-point number, consistent with Python 3.x division behavior, even if both operands are integers. Error handling mechanism ahould check potential issues: if `other` is zero, a `ZeroDivisionError` is raised to prevent division by zero. Additionally, if `other` is not a scalar type (int or float), a `TypeError` is raised to enforce the type requirement for the scalar divisor.
6. `__add__(self, other: Union[gergen, int, float])` → `gergen`: This method facilitates element-wise addition between two `gergen` instances or between a `gergen` and a scalar (int/float). For `gergen`-to-`gergen` addition, it iterates over corresponding elements from both instances, adding them together. If one operand is a scalar, this value is added to every element within the `gergen` instance. The method performs a dimensionality check when both operands are `gergen` instances to ensure their shapes are compatible for element-wise operations. If the dimensions do not align, a `ValueError` is raised, indicating a mismatch in dimensions. Additionally, if the `other` parameter is of an unsupported type, a `TypeError` is raised to maintain type safety. The outcome of the addition is should be returned in a new `gergen` object.
7. `__sub__(self, other: Union[gergen, int, float])` → `gergen`: This method enables element-wise subtraction, either between two `gergen` instances or between a `gergen` and a scalar (int/float). For `gergen`-to-`gergen` subtraction, corresponding elements from each instance are subtracted. When operating with a scalar, the scalar value is subtracted from each element of the `gergen` instance. The method ensures that dimensions are compatible when both operands are `gergen` instances, raising a `ValueError` if there is a mismatch. If the type of `other` is not supported, a `TypeError` is raised. The outcome of the subtraction is should be returned in a new `gergen` object.
8. `uzunluk(self)`: Returns the total number of elements in the `gergen`, represented as a Python integer.
9. `boyut(self)`: Returns the dimensions of the `gergen` object as a Python tuple.
10. `devrik(self)`: Returns the transpose of the `gergen` object as another `gergen` object.
11. `sin(self) -> gergen`: Takes a `gergen` object and applies the sine function to each of its elements. This operation is performed element-wise and returns in a new `gergen` where each value is the sine of the corresponding element in the input `gergen`.
12. `cos(self) -> gergen`: Takes a `gergen` object and applies the cosine function to each of its elements, similar to `sin`.
13. `tan(self) -> gergen`: Takes a `gergen` and applies the tan function to each of its elements.

14. `us(self, n: int) -> gergen`: Takes power by `n` for each element of a given tensor to a specified power `n`. If n is negative, an error will be raised to indicate n should be an integer.

15. `log(self) -> gergen`: Calculates the logarithm of each element of the provided `gergen` with base 10.

16. `ln(self) -> gergen`: Calculates the logarithm to the base `e` for each element of the `gergen`.

17. `L1(self) -> float`: Calculates the L1 norm. The L1 norm, also known as the Manhattan norm, is the sum of the absolute values of the elements in the tensor. For n-dimensional tensor A, L1 norm can be calculated as:

$$\|A\|_1 = \sum_{i_1=1}^{d_1} \sum_{i_2=1}^{d_2} \cdots \sum_{i_n=1}^{d_n} |a_{i_1 i_2 \ldots i_n}|.$$

18. `L2(self) -> float`: Calculates the L2 norm or the Euclidean norm, which is the square root of the sum of the squares of the tensor's elements. For n-dimensional tensor A, L2 norm can be calculated as:

$$\|A\|_2 = \sqrt{\sum_{i_1=1}^{d_1} \sum_{i_2=1}^{d_2} \cdots \sum_{i_n=1}^{d_n} a_{i_1 i_2 \ldots i_n}^2}.$$

19. `Lp(self, p: int) -> float`: Calculates Lp norm, which is general version of L1 and L2 norms by calculating each element to the power of p, summing these values, and then taking the p-th root of the result. For n-dimensional tensor A, Lp norm can be calculated as:

$$\|A\|_p = \left(\sum_{i_1=1}^{d_1} \sum_{i_2=1}^{d_2} \cdots \sum_{i_n=1}^{d_n} |a_{i_1 i_2 \ldots i_n}|^p \right)^{\frac{1}{p}}.$$

    Error Handling:

    – If p is negative, an error will be raised to indicate p should be positive.

20. `listeye(self) -> list`: Converts the `gergen`'s data into a standard Python list structure.

21. `duzlestir(self) -> gergen`: Converts the `gergen`'s multi-dimensional structure into a 1D structure, effectively 'flattening' the tensor. This method does not take any arguments and reorganizes the elements of the `gergen` into a single, continuous linear sequence. For instance, if the `gergen` represents a 2D matrix or a 3D tensor, duzlestir will rearrange its elements into a 1D vector, where the original hierarchical indexing is replaced by a single linear index. The total number of elements remains unchanged, preserving the data while altering its shape to a single dimension.

22. `boyutlandir(self, yeni_boyut: tuple) -> gergen`: Allows for the reorganization of the `gergen`'s structure without altering the underlying data. This method takes a single argument, `yeni_boyut`, which is a tuple specifying the new dimensions that the `gergen` should be reshaped into. When you reshape a tensor, you are changing how its elements are indexed without changing the elements themselves. For example, if you have a 1D tensor (a vector) with 6 elements, you could reshape it into a 2D tensor (a matrix) with dimensions 2×3, 3×2, or even a 3D tensor with dimensions 1×2×3, as long as the total number of elements remains constant. The

new shape must be compatible with the number of elements in the tensor, meaning the product of the dimensions in `yeni_boyut` must equal the total number of elements in the original tensor.

Error Handling:

- If `yeni_boyut` is not provided as a tuple, an error will be raised to indicate the expected format of the new dimensions.
- An error is raised if the product of the new dimensions (`yeni_boyut`) does not match the total number of elements in the original tensor. This check ensures that reshaping is feasible without altering the total data content.

23. `ic_carpim(self, other: gergen) -> float or gergen`: Executes the inner product operation between two `gergen` objects. This method adheres to the mathematical definition of the inner product, which requires both operands to be of the same dimension.
    - **For 1-D Tensors**: When both `gergen` objects represent 1-D tensors, the inner product is calculated as
    $$\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i,$$
    with the constraint that
    $$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n,$$
    meaning both tensors must have the same dimensionality.
    - **Matrix Product Representation**: In cases where the `gergen` objects are treated as column vectors, the inner product can be expressed through the matrix product
    $$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y},$$
    where $\mathbf{x}^T$ denotes the transpose of vector $\mathbf{x}$.
    - **Error Handling**:
        * If either operand is not a `gergen` object, an error is raised to maintain type consistency, ensuring that the operation is performed between two `gergen` instances.
        * For 1-D tensors, if the lengths of the vectors do not match, an error is thrown, emphasizing the requirement for equal dimensionality in the inner product computation.
        * In the case of 2-D tensors, if the number of columns in the first matrix does not equal the number of rows in the second, an error is raised, reflecting the necessity for compatible dimensions in matrix multiplication.

24. `dis_carpim(self, other)`: Compute the outer product of two vectors. The outer product of two vectors $\mathbf{x}$ and $\mathbf{y}$, denoted as $\mathbf{x} \otimes \mathbf{y}$ or $\mathbf{x}\mathbf{y}^T$, is defined as a matrix where each element (i, j) is the product of the i-th element of $\mathbf{x}$ and the j-th element of $\mathbf{y}$. This operation is represented as:
$$\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \dots & x_1 y_m \\ x_2 y_1 & x_2 y_2 & \dots & x_2 y_m \\ \vdots & \vdots & \ddots & \vdots \\ x_n y_1 & x_n y_2 & \dots & x_n y_m \end{bmatrix},$$
where $\mathbf{x}$ is an n-dimensional vector and $\mathbf{y}$ is an m-dimensional vector. The resulting matrix from the outer product has dimensions $n \times m$.

Error handling:

- Verifying that the `other` parameter is indeed a `gergen` object. If not, the method raises a `TypeError` with a message indicating that both operands must be `gergen` instances.
- Ensuring that both `self.veri` and `other.veri` are indeed vectors (1-D arrays). If either operand is not a vector, the method raises a `ValueError` stating that both operands must be 1-D arrays to compute the outer product.

25. `topla(self, eksen=None) -> float/gergen`: Adds up values in `gergen`. If `eksen` is `None`, all elements are added. If `eksen` is not `None`, you can see the examples below:
    - **Column-wise Addition (`eksen=0`):** Elements over the vertical axis are added and returned as a `gergen` with the same size as the number of columns.
    - **Row-wise Addition (`eksen=1`):** Elements over the horizontal axis are added and returned as a `gergen` with the same size as the number of rows.

    Error Handling:
    - If the specified `eksen` is not an integer or `None`, a `TypeError` is raised to indicate that `eksen` must be an integer or `None`.
    - When an `eksen` is provided, the function verifies that it is within the valid range of the data's dimensions. If `eksen` exceeds the dimensions, a `ValueError` is raised indicating that the specified eksen is out of bounds.
26. `ortalama(self, eksen=None) -> float/gergen`: Computes average (mean) of elements in a tensor, with the flexibility to compute this average across different axes of the tensor based on the `eksen` parameter (similar to `topla`).

    When no `eksen` parameter is specified, the method computes the overall average of all elements within the tensor, treating it as a flattened array. This is akin to calculating the mean value of a set of numbers.

    Error Handling:
    - If the specified `eksen` is not an integer or `None`, a `TypeError` is raised to indicate that `eksen` must be an integer or `None`.
    - When an `eksen` is provided, the function verifies that it is within the valid range of the data's dimensions. If `eksen` exceeds the dimensions, a `ValueError` is raised indicating that the specified `eksen` is out of bounds.

2.3 Operation Class Definition

The `Operation` class serves as a base class for operations in the `gergen` class. It is defined as follows:

```
class Operation:
    def __call__(self, *operands):
        self.operands = operands
        self.outputs = None
        return self.ileri(*operands)

    def ileri(self, *operands):
        raise NotImplementedError
```

The `__call__` method allows instances of subclasses of `Operation` to be used as if they were functions, enabling a concise syntax for applying operations to operands. The `ileri` method is intended to be overridden by subclasses to define the specific behavior of the operation.

In the context of the `gergen` class, the `Operation` class serves as a foundational component for defining various mathematical and tensor operations. When creating new operations, such as addition, multiplication, or more complex functions, you should define subclasses of Operation and implement the `ileri` method to encapsulate the operation's specific logic. The `__call__` method in the Operation base class will automatically handle the invocation of the `ileri` method, treating instances of these subclasses as callable objects. To integrate an operation into the `gergen` class, instantiate the corresponding Operation subclass and pass the necessary operands (other `gergen` instances or scalars) to perform the operation, ultimately returning a new `gergen` object that represents the result.

3 COMPARISON WITH NumPy

After completing the gergen class, the next step is to verify your calculations using the corresponding NumPy functions. While doing so, we will also compare the running times of the operations. For given three equations you will compare your results with their NumPy equivalent and report the time difference:

1. Using `rastgele_gercek`, generate two `gergen` objects **A** and **B** with shapes (64, 64) and calculate:

$$\mathbf{A}^T\mathbf{B}.$$

   Then, calculate the same function with NumPy and report the time and difference.

2. Using `rastgele_gercek`, generate three `gergen`s **A**, **B** and **C** with shapes (4,16,16,16) and report the time and result with their NumPy equivalent:

$$(A \times B + C \times A + B \times C).ortalama().$$

3. Using `rastgele_gercek`, generate two `gergen`'s **A** and **B** with shapes (3,64,64) and report the time and result with their NumPy equivalent:

$$\frac{\ln\left((\sin(A) + \cos(B))^2\right)}{8}.$$

Within your Jupyter Notebook, there are three predefined functions: `example_1()`, `example_2()`, and `example_3()`. It is expected that you carry out the necessary comparisons within these functions.

4 GRADING/SPECIFICATIONS

- You will be given a Jupyter Notebook (.ipynb) file that contains the skeleton of the tensor library.
- Your task is to complete the implementation of the provided classes and functions.
- You should use Python 3.6+.
- You should save your class and method implementations as a python file named `cergen.py`.

- You can **NOT** use external libraries such as NumPy, PyTorch, TensorFlow, or any other libraries that provide high-level tensor operations. All implementations must utilize only the standard built-in Python libraries. You can only use NumPy in "Comparison with NumPy" section.
- Ensure that your `gergen` library can handle different scenarios and edge cases, providing correct results and raising errors when appropriate. Your implementations must especially handle shape inconsistencies. Whenever shape mismatches occur in operations (e.g., element-wise operations, dot products, broadcasting), your code should raise an error with an informative error message explaining the nature of the inconsistency. You should include checks to verify that operations are being performed on tensors of compatible shapes as per the operation requirements.
- You can define your own variables and helper functions.