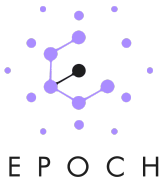


Spring Camp Session 1 Tasks

Hope you all enjoyed session 1 on basic introduction to python libraries and Classes. Here's a few tasks from Epoch. Hope you have fun and feel free to contact anyone from Epoch if you have any questions.

(Note: We will be keeping an eye out for good submissions, so do give your best shot. The submissions don't have to be perfect, we just want to see your enthusiasm and effort)

Submission Form

Task - 0:

Now that you have a good understanding on using numpy, matplotlib etc, try the task1 from the intro session.

Task - 1:

Implement a 4-way kaleidoscope effect using NumPy and Matplotlib.

1. Grayscale Image:
   - Load or generate a grayscale image using NumPy.
   - Apply a 4-way kaleidoscope transformation by reflecting and rotating the quadrants appropriately
   - Use Matplotlib to visualize the transformed image.

2. RGB Image:
   - Extend the implementation to work with RGB images.
   - Ensure that each color channel undergoes the same transformation as the grayscale version.
   - Display the resulting kaleidoscope effect using Matplotlib.

An Example for the RGB image:

Input image

Output image

Task - 2:

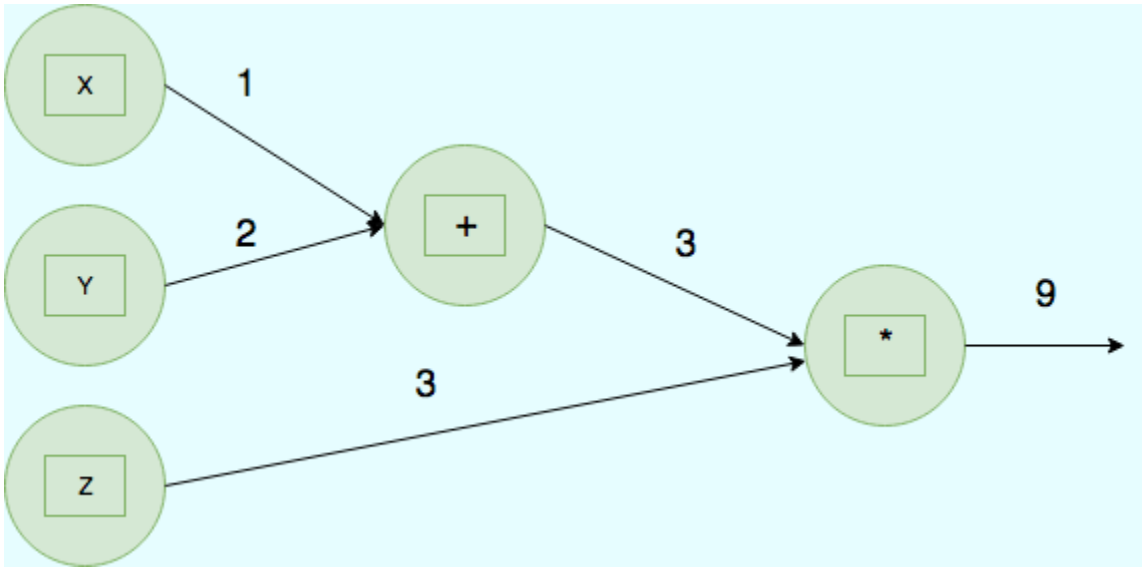
Implement a **`Node`** class to represent values in a **computational graph**. This class should support **element-wise arithmetic operations** and **track dependencies** for automatic differentiation.

- **Define a `Node` class** with the following attributes:
  - **`value`:** Stores the numerical value (NumPy array).
  - **`children`:** Tracks dependencies (other **`Node`** objects used to compute the current node).
  - **`op`:** Stores the operation that created the node (e.g., **`"add"`**, **`"mul"`**).
  - **`op_args`:** Stores additional operation-related constants if needed.

- **Overload arithmetic operators** to support element-wise operations between `Node` objects and constants:
  - Addition (**`+`**)
  - Subtraction (**`-`**)
  - Multiplication (**`*`**)
  - Division (**`/`**)
  - Exponentiation (**`**`**)

- **Track dependencies** in the computation graph:
  - If an operation involves two **`Node`** objects, store both as children.
  - If an operation involves a constant, store only the **`Node`** operand.

Example of a Computational Graph:



Here let a = (x+y)*z be the output and b = x+y.
In the above example,

- b and z are children of a.
- x and y are children of b.

Bonus Task:

Implement a **`backward()`** method for the **`Node`** class to compute **gradients** using the **chain rule** of differentiation. This method should **propagate gradients** correctly through the computation graph from the output node to its dependencies.

1. **Define a `backward()` method** in the **`Node`** class that:
    - Initializes the gradient (**`grad`**) to **1** if the node is the starting point of backpropagation.
    - Propagates gradients **recursively** to all dependent nodes (**children**) based on the operation performed.

2. **Implement gradient calculations** for the following operations:
    - **Addition (+)**: Gradient is passed unchanged to both operands
    - **Subtraction (-)**: Left operand receives **`grad`**, right operand receives **`-grad`**.
    - **Multiplication (*)**: Uses the product rule.
    - **Division (/)**: Uses the quotient rule.
    - **Exponentiation (**)**: Uses the power rule and logarithm differentiation.

3. **Perform backpropagation**:
    - Compute gradients by calling **`backward()`** from the final output node.
    - Ensure gradients **accumulate** correctly when a node is used in multiple operations.

4. **Verify correctness** by performing backpropagation on an example computation:
    - Create two **`Node`** objects (**`a and b`**) initialized with NumPy arrays.
    - Compute intermediate and final results:
        - $c = a \times 2 + b$
        - $d = c^3$
    - Call **`d.backward()`** to compute the gradient of **`d`** with **`a`**, **`b`**, and **`c`**.
    - Print the computed gradients.