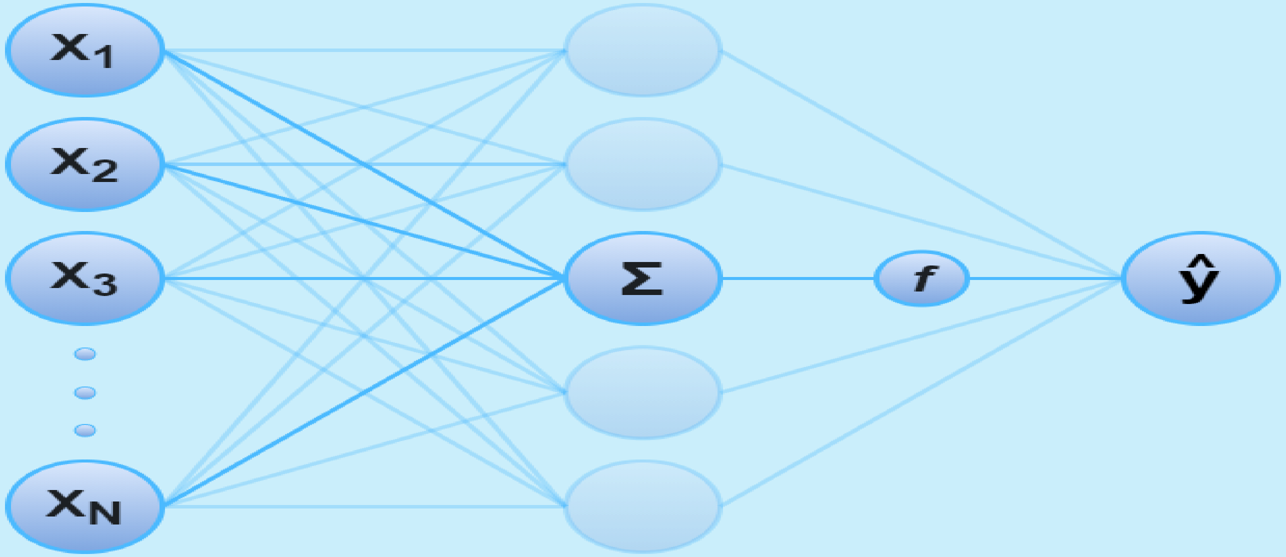
# Automatic Differentiation:

## with Basic Operations

Kishan Chakraborty

246150004

# Motivation



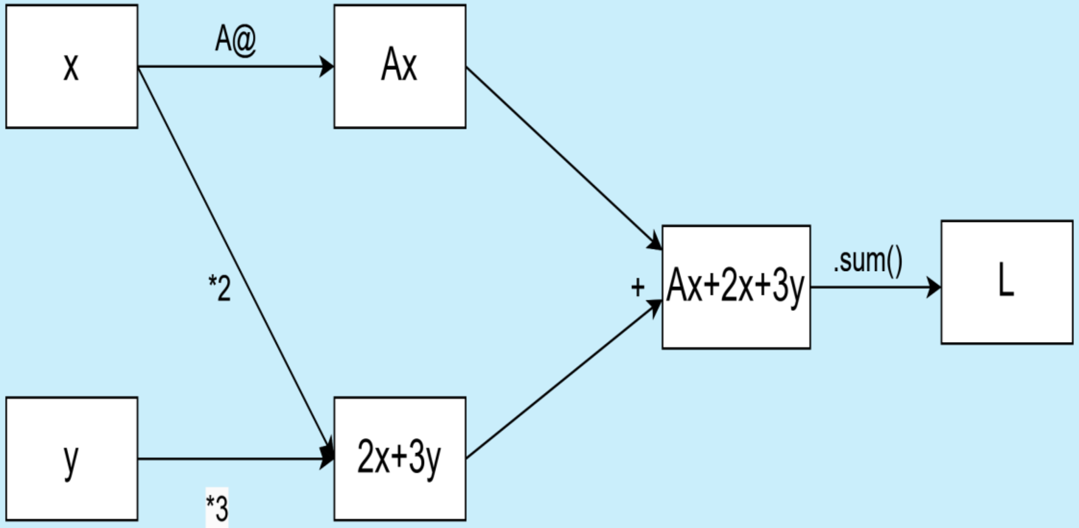
- Mechanics of PyTorch's .backward() method.
  - While training neural networks using PyTorch, .backward() method is used to do the backpropagation (calculating gradients.)

# Goal

- Create our own tensor object and perform basic operations.
  - Implement basic tensor operations from scratch.
  - Handle operation between custom built tensor object and standard objects like int, float, etc.

- Understand how .backward() method works for any tensor.
  - How chain rule works.
  - Building computational graph.
  - Gradient flow in a computational graph.

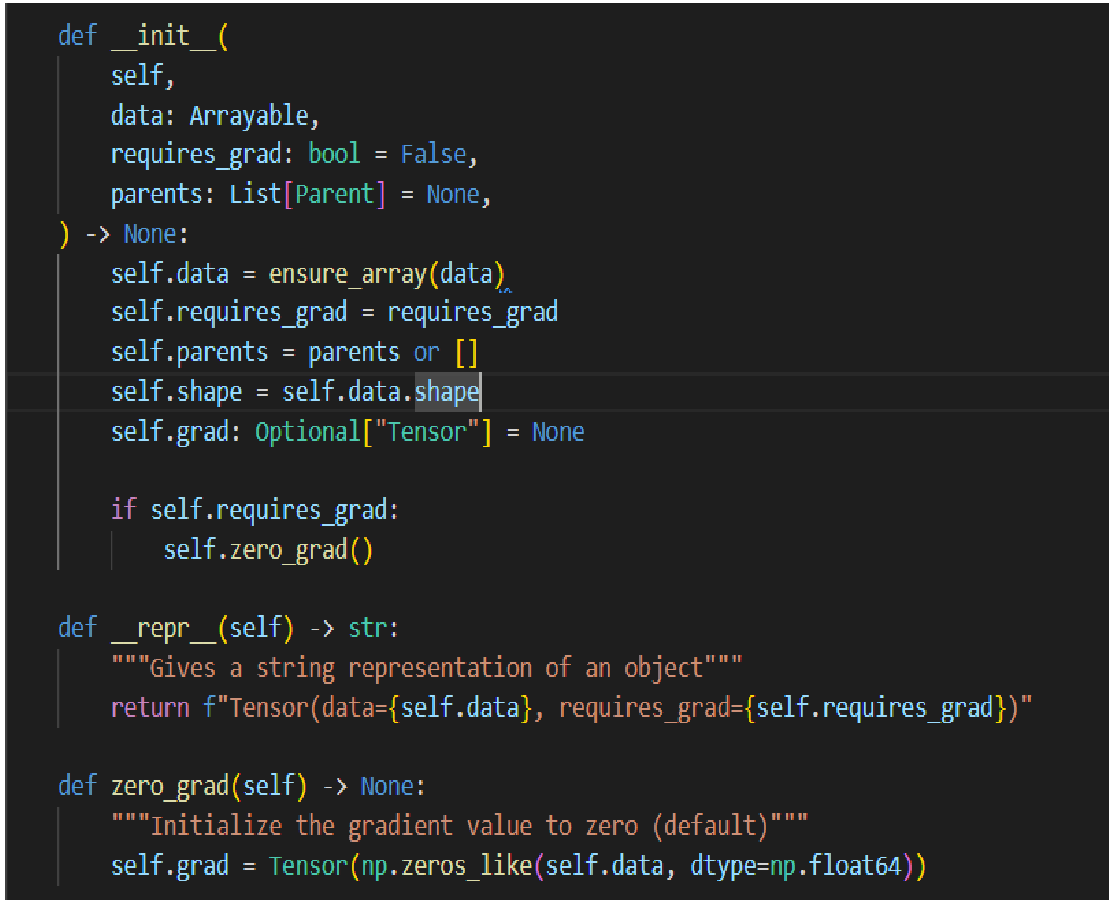
# **Pseudocode1:** Example



- Tensor Class
  - Value
  - Gradient value
  - Information about parents
- Parent Class
  - Tensor object
  - Gradient Function
- Operations
  - Override the standard operations for custom class.

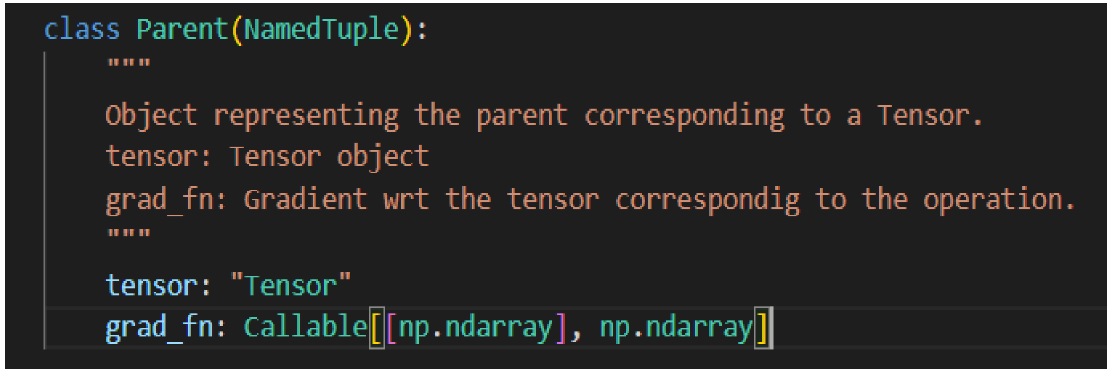
# **Code Snippets1:** Example

- Tensor Class



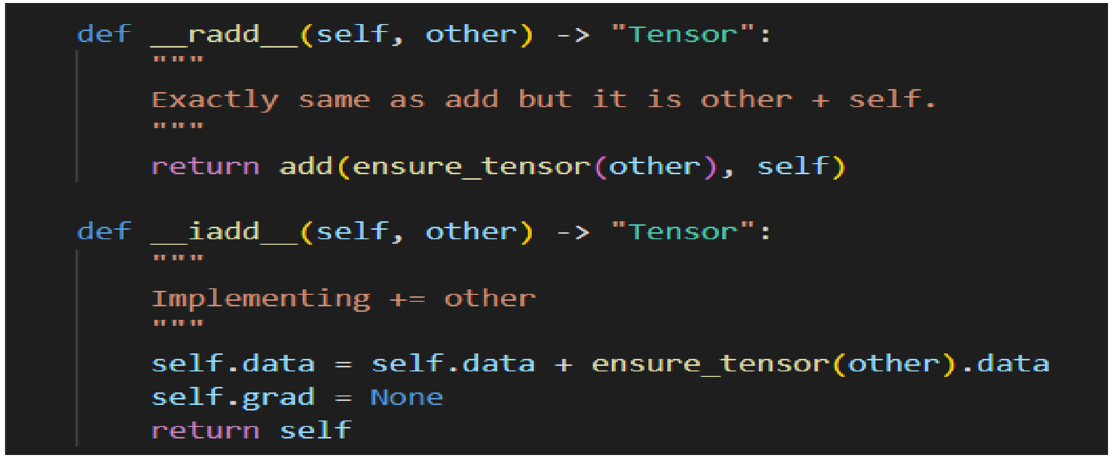
```python
def __init__(
    self,
    data: Arrayable,
    requires_grad: bool = False,
    parents: List[Parent] = None,
) -> None:
    self.data = ensure_array(data)
    self.requires_grad = requires_grad
    self.parents = parents or []
    self.shape = self.data.shape
    self.grad: Optional["Tensor"] = None

    if self.requires_grad:
        self.zero_grad()

def __repr__(self) -> str:
    """Gives a string representation of an object"""
    return f"Tensor(data={self.data}, requires_grad={self.requires_grad})"

def zero_grad(self) -> None:
    """Initialize the gradient value to zero (default)"""
    self.grad = Tensor(np.zeros_like(self.data, dtype=np.float64))
```

- Parent Class



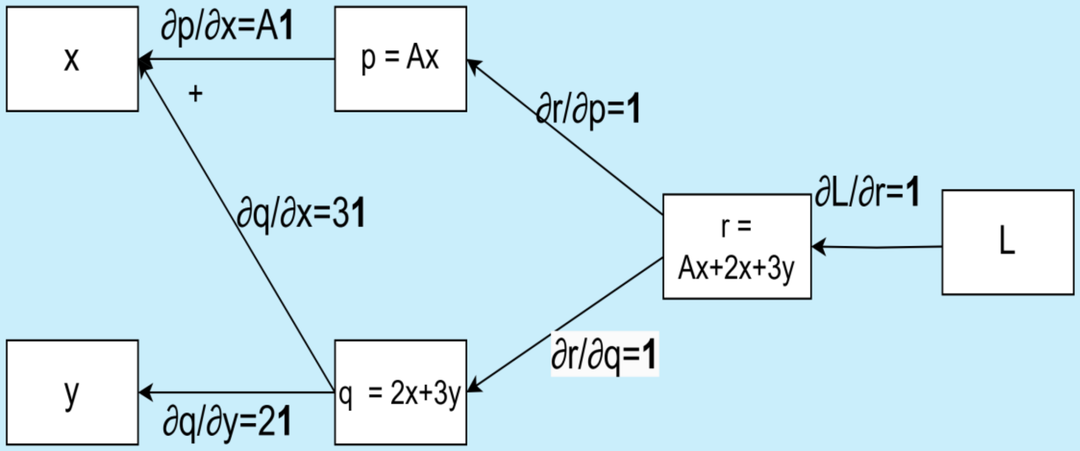
```python
class Parent(NamedTuple):
    """
    Object representing the parent corresponding to a Tensor.
    tensor: Tensor object
    grad_fn: Gradient wrt the tensor correspondig to the operation.
    """
    tensor: "Tensor"
    grad_fn: Callable[[np.ndarray], np.ndarray]
```

- Override Operations



```python
def __radd__(self, other) -> "Tensor":
    """
    Exactly same as add but it is other + self.
    """
    return add(ensure_tensor(other), self)

def __iadd__(self, other) -> "Tensor":
    """
    Implementing += other
    """
    self.data = self.data + ensure_tensor(other).data
    self.grad = None
    return self
```

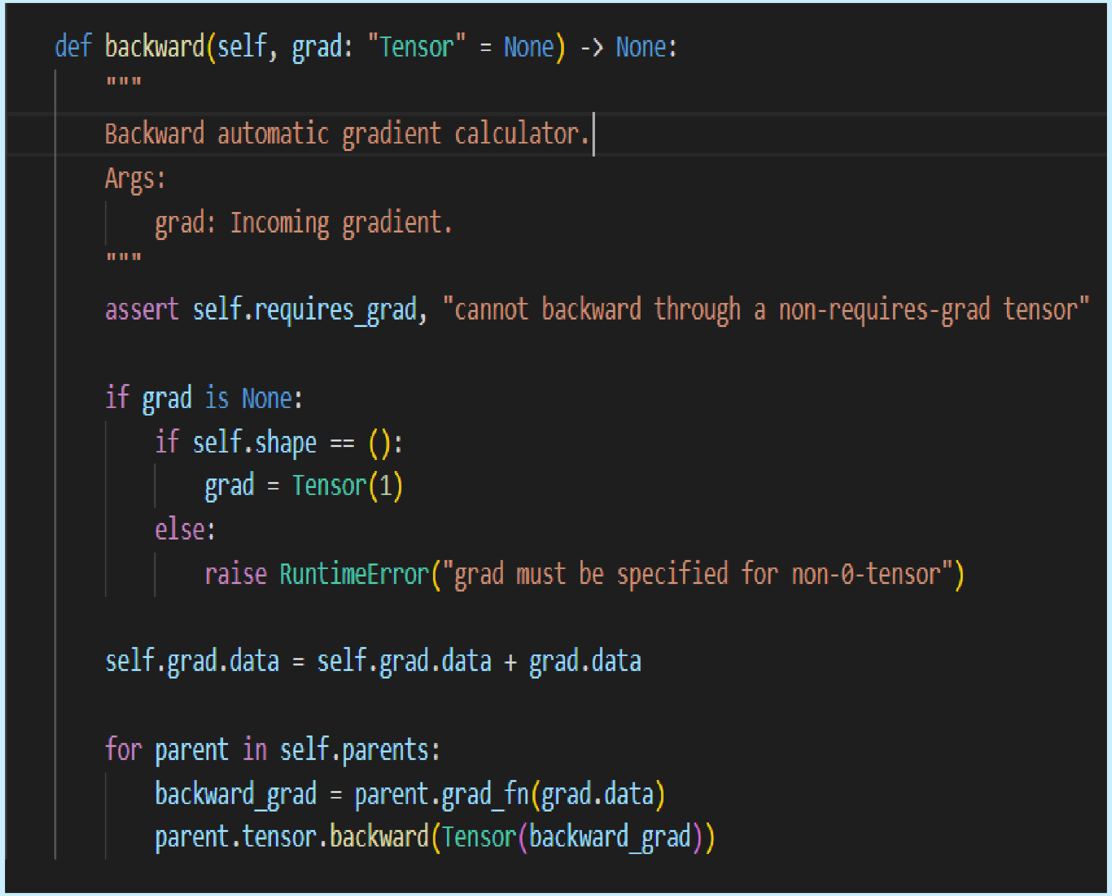
# **Pseudocode2:** Example



- Tensor.backward(grad)
    - Tensor.grad += grad
    - Iterate over all the immediate parents
        - grad_parent = parent.grad(grad)
        - parent.backward(grad_parent)
- Tensor.grad()
    - Calculate gradient using function based on the operation
        - The function is stored as a method of the parent class during a specific operation.
        - Consider the incoming gradient.
        - Depend upon operation in which tensor was part of.

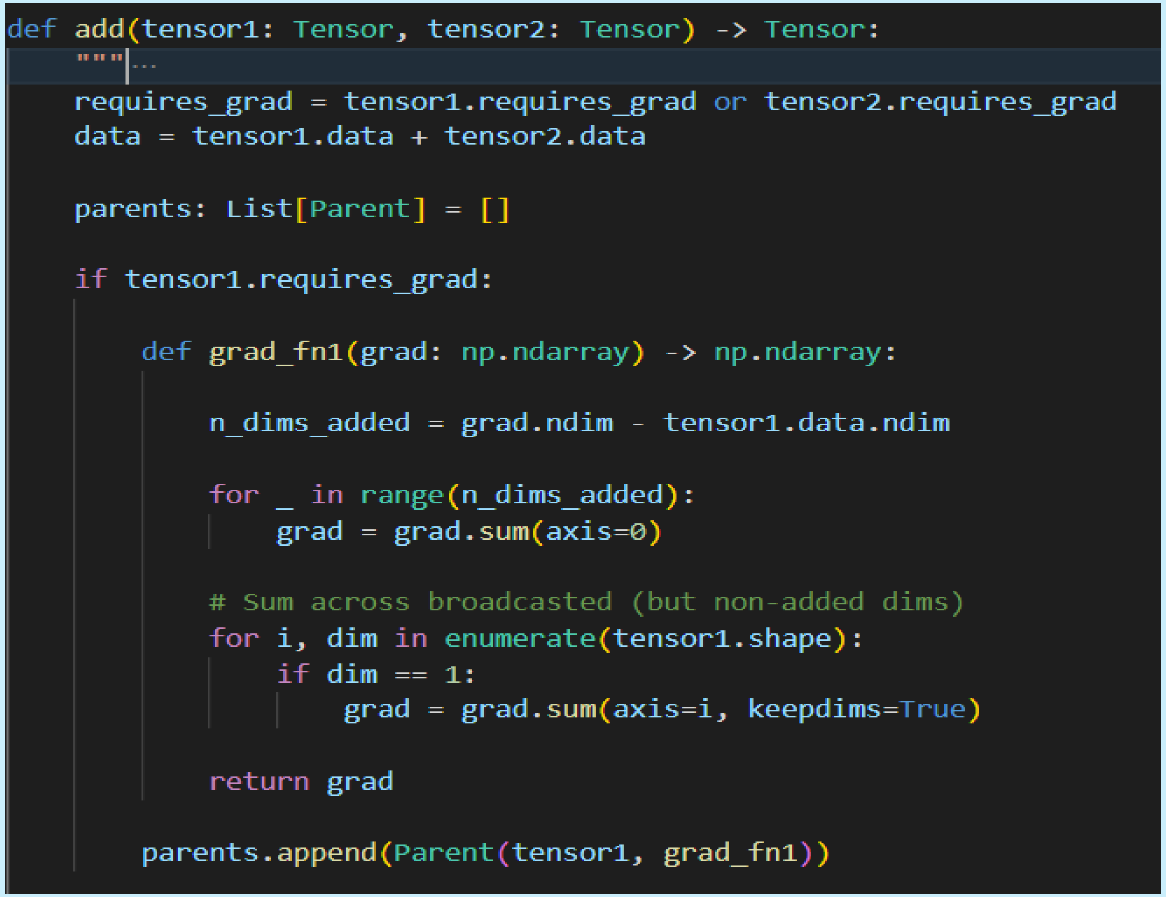
# **Code Snippets1**: Example

- .Backward()



```
def backward(self, grad: "Tensor" = None) -> None:
    """
    Backward automatic gradient calculator.
    Args:
        grad: Incoming gradient.
    """
    assert self.requires_grad, "cannot backward through a non-requires-grad tensor"

    if grad is None:
        if self.shape == ():
            grad = Tensor(1)
        else:
            raise RuntimeError("grad must be specified for non-0-tensor")

    self.grad.data = self.grad.data + grad.data

    for parent in self.parents:
        backward_grad = parent.grad_fn(grad.data)
        parent.tensor.backward(Tensor(backward_grad))
```

- Operation and grad function.



```
def add(tensor1: Tensor, tensor2: Tensor) -> Tensor:
    """...
    requires_grad = tensor1.requires_grad or tensor2.requires_grad
    data = tensor1.data + tensor2.data

    parents: List[Parent] = []

    if tensor1.requires_grad:

        def grad_fn1(grad: np.ndarray) -> np.ndarray:

            n_dims_added = grad.ndim - tensor1.data.ndim

            for _ in range(n_dims_added):
                grad = grad.sum(axis=0)

            # Sum across broadcasted (but non-added dims)
            for i, dim in enumerate(tensor1.shape):
                if dim == 1:
                    grad = grad.sum(axis=i, keepdims=True)

            return grad

        parents.append(Parent(tensor1, grad_fn1))
```

# **Learnings:** Python

- Operations(symbols) can be overriden.
  - Standard mathematical symbols can also be defined for custom objects.
- How operations are defined in python.
  - There is a difference between "Tensor" + 3 and 3 + "Tensor".
- Testing modules
  - Used unittest modules to write testcases for different operations.
- Local module installation
  - For fluent implementation of module imports from sibling directories use settup.py module for local installation of AutoGrad package.

# **Learnings:** Git

- Use of different branch for collaboration.
- Merging of different branches for final compilation.

Thank You !!