Name: Adithya M   SRN: PES1UG20CS621   Section: K

Week 5

```
import numpy as np
class Tensor:

    """
    Tensor Wrapper for Numpy arrays.
    Implements some binary operators.
    Array Broadcasting is disabled
    Args:
        arr: Numpy array (numerical (int, float))
        requires_grad: If the tensor requires_grad (bool)(otherwise gradient dont apply to the tensor)
    """

    def __init__(self, arr, requires_grad=True):

        self.arr = arr
        self.requires_grad = requires_grad

        # When node is created without predecessor the op is denoted as 'leaf'
        # 'leaf' signifies leaf node
        self.history = ["leaf", None, None]
        # History stores the information of the operation that created the Tensor.
        # Check set_history

        # Gradient of the tensor
        self.zero_grad()
        self.shape = self.arr.shape
```

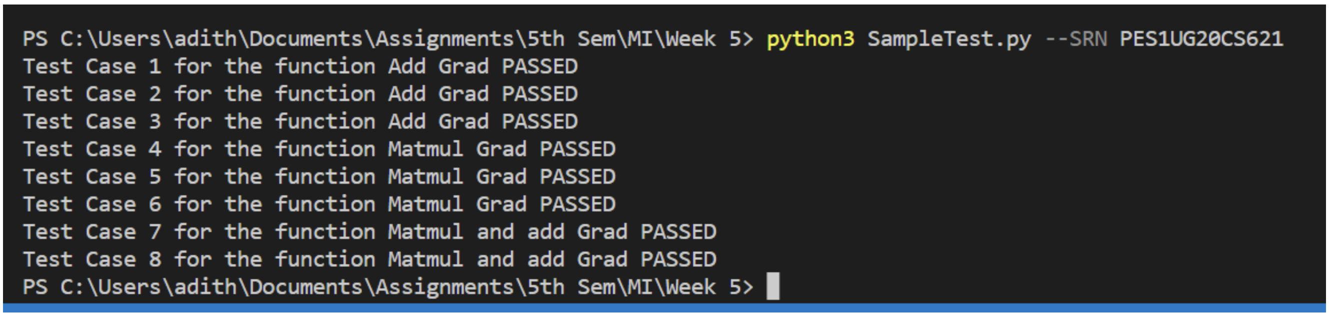
```
def zero_grad(self):
    """
    Set grad to zero
    """
    self.grad = np.zeros_like(self.arr)

def set_history(self, op, operand1, operand2):
    """
    Set History of the node, indicating how the node was created.
    Ex:-
        history -> ['add', operand1(tensor), operand2(tensor)]
        history -> ['leaf', None, None] if tensor created directly
    Args:
        op: {'add', 'sub', 'mul', 'pow', 'matmul', 'leaf') (str)
        operand1: First operand to the operator. (Tensor object)
        operand2: Second operand to the operator. (Tensor object)
    """
    self.history = []
    self.history.append(op)
    self.requires_grad = False
    self.history.append(operand1)
    self.history.append(operand2)

    if operand1.requires_grad or operand2.requires_grad:
        self.requires_grad = True

"""
Addition Operation
Tensor-Tensor(Element Wise)
__add__: Invoked when left operand of + is Tensor
grad_add: Gradient computation through the add operation
```

```
"""


def __add__(self, other):
   """
   Args:
      other: The second operand.(Tensor)
            Ex: a+b then other -> b, self -> a
   Returns:
      Tensor: That contains the result of operation
   """
   if isinstance(other, self.__class__):
      if self.shape != other.shape:
         raise ArithmeticError(
            f"Shape mismatch for +: '{self.shape}' and '{other.shape}' "
         )
      out = self.arr + other.arr
      out_tensor = Tensor(out)
      out_tensor.set_history("add", self, other)

   else:
      raise TypeError(
         f"unsupported operand type(s) for +: '{self.__class__}' and '{type(other)}'"
      )

   return out_tensor

"""
Matrix Multiplication Operation (@)
Tensor-Tensor
__matmul__: Invoked when left operand of @ is Tensor
grad_matmul: Gradient computation through the matrix multiplication operation
```

```
"""

def __matmul__(self, other):
    """
    Args:
        other: The second operand.(Tensor)
                Ex: a+b then other -> b, self -> a
    Returns:
        Tensor: That contains the result of operation
    """
    if not isinstance(other, self.__class__):
        raise TypeError(
            f"unsupported operand type(s) for matmul: '{self.__class__}' and '{type(other)}'"
        )
    if self.shape[-1] != other.shape[-2]:
        raise ArithmeticError(
            f"Shape mismatch for matmul: '{self.shape}' and '{other.shape}' "
        )
    out = self.arr @ other.arr
    out_tensor = Tensor(out)
    out_tensor.set_history("matmul", self, other)

    return out_tensor

def grad_add(self, gradients=None):
    """
    Find gradients through add operation
    gradients: Gradients from successing operation. (numpy float/int)
    Returns:
        Tuple: (grad1, grad2)
        grad1: Numpy Matrix or Vector(float/int) -> Represents gradients passed to first operand
```

```
            grad2: Numpy Matrix or Vector(float/int) -> Represents gradients passed to second operand
            Ex:
                c = a+b
                Gradient to a and b
        """
        # TODO
        op1 = self.history[1]
        op2 = self.history[2]
        op1.grad = np.zeros_like(op1.arr)
        op2.grad = np.zeros_like(op2.arr)
        if op1.requires_grad:
            op1.grad += np.ones_like(op1.arr)
        if op2.requires_grad:
            op2.grad += np.ones_like(op2.arr)
        if gradients is None:
            return (op1.grad, op2.grad)
        if op1.requires_grad:
            op1.grad = np.multiply(np.ones_like(op1.arr), gradients)
        if op2.requires_grad:
            op2.grad = np.multiply(np.ones_like(op2.arr), gradients)
        return (op1.grad, op2.grad)

    def grad_matmul(self, gradients=None):
        """
        Find gradients through matmul operation
        gradients: Gradients from successing operation. (numpy float/int)
        Returns:
            Tuple: (grad1, grad2)
            grad1: Numpy Matrix or Vector(float/int) -> Represents gradients passed to first operand
            grad2: Numpy Matrix or Vector(float/int) -> Represents gradients passed to second operand
            Ex:
                c = a@b

                Gradients to a and b

        """

        # TODO

        op1 = self.history[1]

        op2 = self.history[2]

        if gradients is None:

            if op1.requires_grad:

                op1.grad += np.matmul(np.ones_like(op1.arr), op2.arr.transpose())

            if op2.requires_grad:

                op2.grad += (np.matmul(np.ones_like(op2.arr), op1.arr)).transpose()

        else:

            if op1.requires_grad:

                op1.grad += np.multiply(

                    np.matmul(np.ones_like(op1.arr), op2.arr.transpose()), gradients

                )

            if op2.requires_grad:

                op2.grad += np.multiply(

                    np.matmul(np.ones_like(op2.arr), op1.arr).transpose(), gradients

                )


        return (op1.grad, op2.grad)


    def backward(self, gradients=None):

        """

        Backward Pass until leaf node.

        Setting the gradient of which is the partial derivative of node(Tensor)

        the backward in called on wrt to the leaf node(Tensor).

        Ex:

            a = Tensor(..) #leaf

            b = Tensor(..) #leaf
            c = a+b
            c.backward()
            computes:
                dc/da -> Store in a.grad if a requires_grad
                dc/db -> Store in b.grad if b requires_grad
        Args:
            gradients: Gradients passed from succeeding node
        Returns:
            Nothing. (The gradients of leaf have to set in their respective attribute(leafobj.grad))
        """
        # TODO

        if self.requires_grad == None:
            return
        if self.history[0] == "add":

            gradient = self.grad_add(gradients)
            if self.history[1]:
                self.history[1].backward(gradient[0])
            if self.history[2]:
                self.history[2].backward(gradient[1])

        elif self.history[0] == "matmul":
            gradient = self.grad_matmul(gradients)
            if self.history[1]:
                self.history[1].backward(gradient[0])
            if self.history[2]:
                self.history[2].backward(gradient[1])
        else:
            if self.requires_grad:
                self.grad = gradients
```

OUTPUT:



```
PS C:\Users\adith\Documents\Assignments\5th Sem\MI\Week 5> python3 SampleTest.py --SRN PES1UG20CS621
Test Case 1 for the function Add Grad PASSED
Test Case 2 for the function Add Grad PASSED
Test Case 3 for the function Add Grad PASSED
Test Case 4 for the function Matmul Grad PASSED
Test Case 5 for the function Matmul Grad PASSED
Test Case 6 for the function Matmul Grad PASSED
Test Case 7 for the function Matmul and add Grad PASSED
Test Case 8 for the function Matmul and add Grad PASSED
PS C:\Users\adith\Documents\Assignments\5th Sem\MI\Week 5> 
```