**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:

