Week 5 – Artificial Neural Networks

MACHINE INTELLIGENCE LABORATORY- PESU UE19CS305

Teaching Assistants-

sarasharish2000@gmail.com

abaksy@gmail.com

vighneshkamath43@gmail.com

In this week's experiment, you will be mimicking one of the most popular Python frameworks used for deep learning, TensorFlow. TensorFlow stores all operations internally in a graph data structure called a "computation graph". A computation graph is a Directed Acyclic Graph (DAG) in which each node represents a data value (in our case, a multidimensional array called a Tensor) and edges of the graph represent the flow of data through binary operations that is performed on 2 input Tensors and returns a single output Tensor. (**Note the operations in the below diagrams are represented as nodes, this is just for your understanding. In our implementation we will not create a node for an operation).**

Your task in this week's experiment is to write functions that, given a computation graph, compute the gradient of a Tensor variable (say 'x') with respect to the leaf nodes of the computation graph that created the Tensor 'x'. That is to implement '**chain rule'** using computation graphs

For the purposes of this week's experiment, the only operations that can be carried out on 2 Tensors are tensor addition and tensor multiplication (these are the same as matrix addition and matrix multiplication respectively).

You are provided with the following files:

1. **week5.py**
2. **SampleTest.py**

Note: These sample test cases are just for your reference.

Computation Graphs

A computation graph is a Directed Acyclic Graph that represents an ordering of operations between Tensor values. Each node of the computation graph is a data value (in our case, this is an object of the class 'Tensor' which you will implement).(**Note the operations in the below diagrams are represented as nodes, this is just for your understanding. In our implementation we will not create a node for an operation).**
Computations graphs are how chain rule is implemented

For example, assuming the arithmetic operation $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ where *a*, *b*, and *c* are Tensor objects, the computation graph for this operation is represented as:

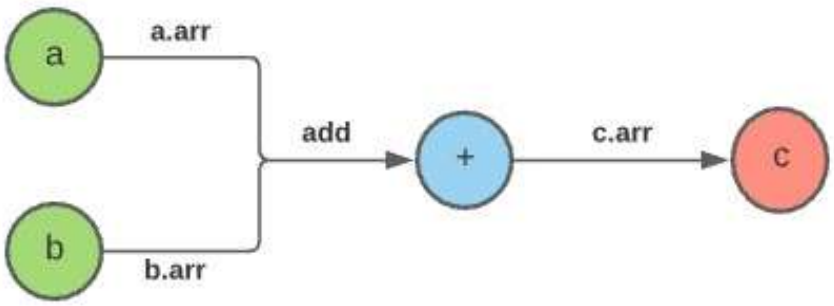

Fig 1: Computation graph for Tensor addition (c = a + b)

For example, assuming the arithmetic operation $\boldsymbol{c} = \boldsymbol{a}@\boldsymbol{b}$ where *a*, *b*, and *c* are Tensor objects and the '@' symbol represents matrix multiplication , the computation graph for this operation is represented as

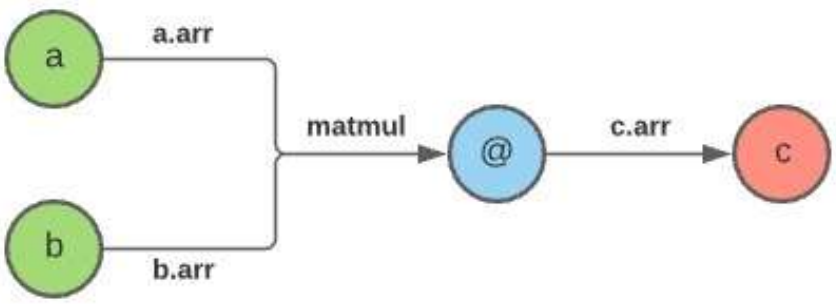

Fig 2: Computation graph for Tensor multiplication (c = a @ b)

Let's look at a more complicated expression, say $d = (a + b)@c$ and its computation graph.

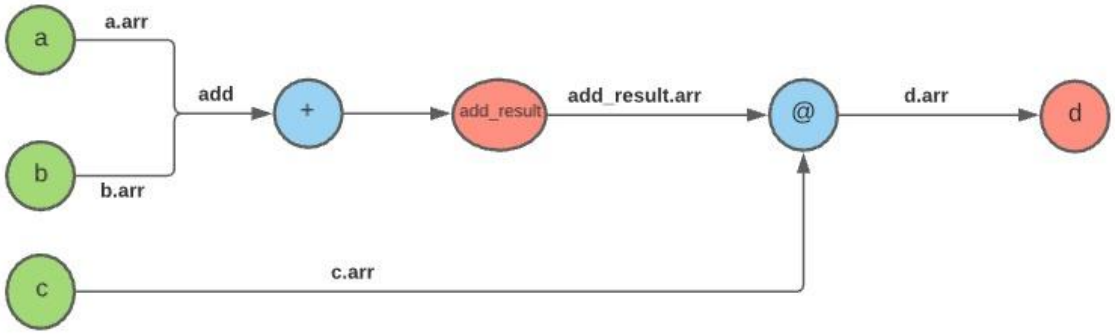

Fig 3: Computation graph for multiple Tensor operations (d = (a+b)@c)

In all 3 diagrams, the nodes coloured green are the **leaf nodes**, nodes in blue represent Tensor operations, and nodes in red (**non-leaf nodes**) represent results of operations carried out on 2 input Tensors.

Your aim in this experiment is to compute the gradient of a leaf node which is the partial derivative of the node where backward is called with respect to the leaf nodes in the computation graph.

For this, you must make use of the chain rule. Let us take the example of $c = (a + b)$. A call to the backward() method of the object 'c' will cause the following flow of operations to take place. Read the below diagram from right to left, following the arrows to understand the operations that take place.

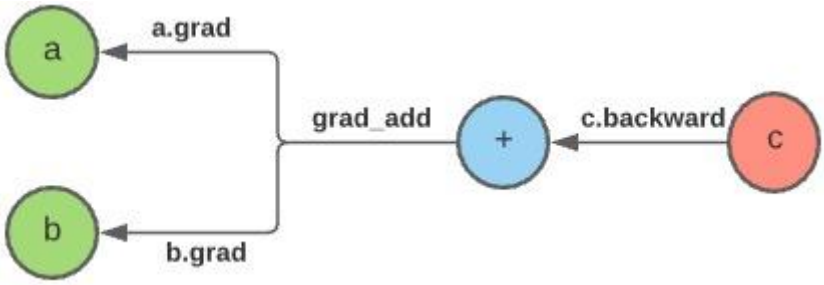

Fig 4: Computation graph for gradient calculation of 'c' w.r.t 'a' and 'b'

Calling c.backward() will lead to

a.grad $\leftarrow \frac{\delta c}{\delta a}$

b.grad $\leftarrow \frac{\delta c}{\delta b}$

('grad' is an attribute of Tensor object)

c.grad remain unchanged since it is not a leaf node

Computation Graph Implementation

For this lab assignment, the following is already implemented for you:

1) The addition operation between 2 Tensors

2) The matmul operation between 2 Tensors

3) Creating the computation graph whenever an operation takes place

IMPORTANT:

The computation graph is implemented using the **History data structure**. Each Tensor object in the graph has a **history attribute**. **Setting the history attribute of node has already been implemented, you will only need to understand its structure as shown below**. Have a clear understanding of the history attribute before you start writing the code.

The history is a list of length 3 that contains, in order:

1) **history[0]:** A **string** that can either be valued as **"leaf", "add" or "matmul".**

- A value of **"leaf"** indicates that the Tensor object was created from scratch and not as the result of any arithmetic operation. All objects created from the constructor (__init__) are leaf nodes by default
- A value of **"matmul"** or **"add"** indicates that the Tensor object was created using either a matrix multiplication or addition operation respectively.

2) **history[1] and history[2]:** A reference to the left and right operands respectively that, when operated with the operand history[0] resulted in the creation of the current Tensor object. These are objects of type **Tensor** for a non-leaf node. In the case of a **leaf** node, these are of **None** type**.(Note this does not contain a copy of the operands but the operands themselves "a reference")**

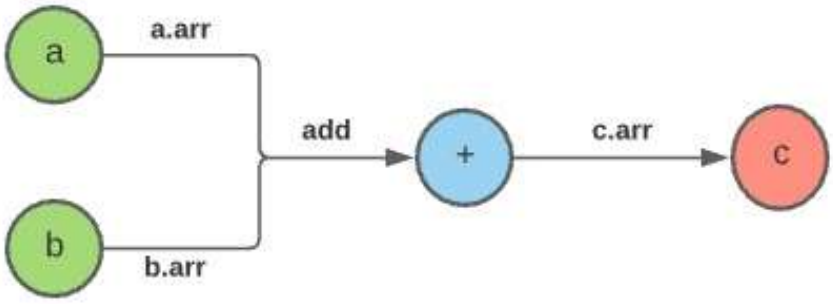

Fig 1: Computation graph of (c = a + b)

For example, refer to the above. The history of **'a'** and **'b'** would be the list **['leaf', None, None]** as they are both leaf nodes. Now, as soon as the node **'c'** is created, its history would be the list **['add', a, b]** where 'a' and 'b' are references to the Tensor objects 'a' and 'b'.

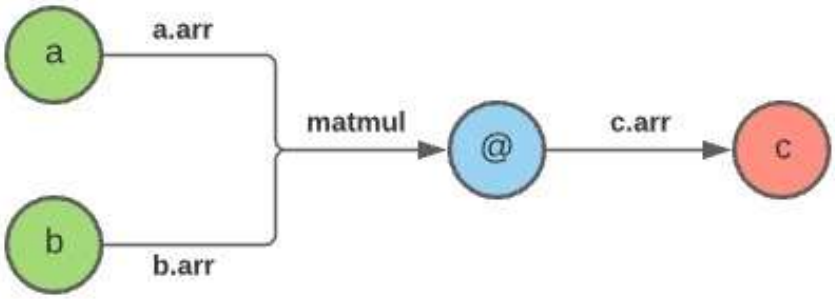

Fig 2: Computation graph of (c = a@b)

In the figure above the history of **'a'** and **'b'** would be the list **['leaf', None, None]** as they are both leaf nodes. Now, as soon as the node **'c'** is created, its history would be the list **['matmul', a, b]** where 'a' and 'b' are references to the Tensor objects 'a' and 'b'.

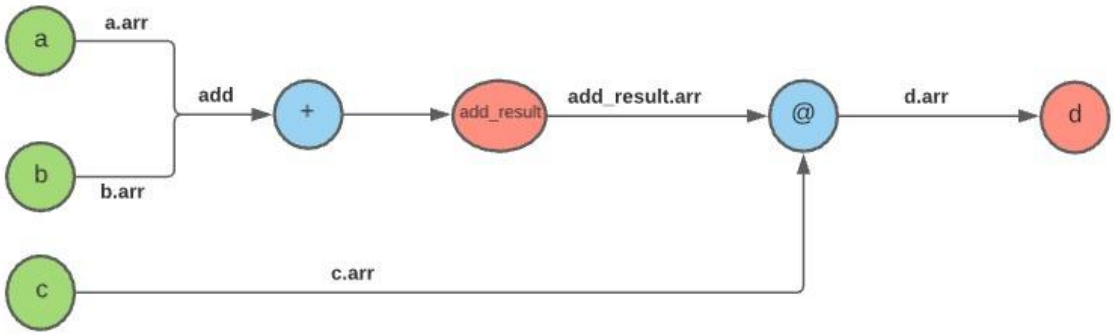

Fig 3: Computation graph for multiple Tensor operations (d = (a+b)@c)

In the figure above the history of **'a'** ,**'b'** and **'c'** would be the list **['leaf', None, None]** as they are both leaf nodes. After the first operation **'add_result'** is created with a history list **['add', a, b]** . **Note 'add_result' is not explicitly present as a variable but after addition completes the matmul operation receives the 'add_result' as an operand.** After the matmul operation completes the node 'd' will have a history list **['matmul', add_result, c].** **'add_result' is present in node d's history even though it isn't explicitly present as a variable.**

Calling d.backward() will lead to

a.grad $\leftarrow \frac{\delta d}{\delta a} = \frac{\delta\, d}{\delta\, add_{result}} \times \frac{\delta\, add_{result}}{\delta\, a}$, b.grad $\leftarrow \frac{\delta d}{\delta b} = \frac{\delta\, d}{\delta\, add_{result}} \times \frac{\delta\, add_{result}}{\delta\, b}$ (chain rule)

c.grad $\leftarrow \frac{\delta d}{\delta c}$

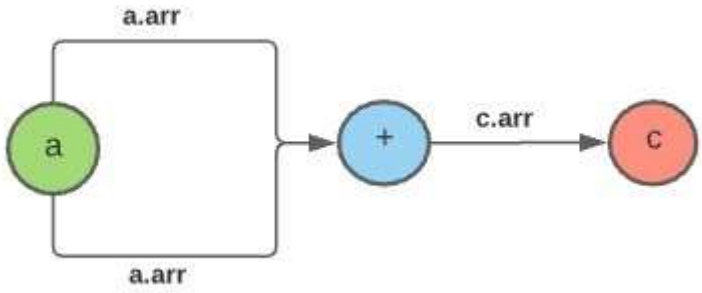

Fig: 5 Sample Case: c = a+a (history of c will contain ['add', a, a])

Important: **The computation graph is present internally(with history), that is a node holds reference to its parents(the nodes they are derived from) and the parents hold reference to their own parents and so on... All non-leaf nodes are guaranteed to have a reference to their parents in history.**

NOTE: Think about how one would traverse the computation graph starting from the result node to the leaf nodes.
Hint: Think about the direction in which the operation flows and the direction in which the gradients flow from Figure 1 and Figure 4.

Important Points:

1. Please do not make changes to the function definitions that are provided to you, as well as the functions that have **already been implemented**. Use the skeleton as it has been given. Also do not make changes to the sample test file provided to you.
2. You are free to write any helper functions that can be called in any of these predefined functions given to you. Helper functions must be only in the file named 'YOUR_SRN.py'.
3. Your code will be auto evaluated by our testing script and our dataset and test cases will not be revealed. Please ensure you take care of all edge cases!
4. **The experiment is subject to zero tolerance for plagiarism. Your code will be tested for plagiarism against every code of all the sections and if found plagiarized both the receiver and provider will get zero marks without any scope for explanation.**
5. **Kindly do not change variable names or use any other techniques to escape from plagiarism, as the plagiarism checker is able to catch such plagiarism**
6. Hidden test cases will not be revealed post evaluation.
7. **The code for performing the operations, setting the history(building the graph) has already been implemented.**
8. **The gradient of a Tensor should always be the same shape as that of the tensor array itself.**
9. All operations are performed between Tensor objects only. **The result of an operation will never overwrite any of the previous operands. Ex: a = a+b or a += b or (c = a+b, a = c+b) will never occur in any of the tests(Not Valid and need not be handled).**
10. All shape mismatches that may occur during addition and matrix multiplication have been handled already using exceptions. You are **not required** to handle these any further. You need not handle any numpy broadcasting.(All Tensor additions are guaranteed to be performed on Tensors with same shape i.e (n x n)+(n x n) and Matmul is guaranteed to be performed on Tensors with compatible shapes like (m x n)@(n x p))
11. Each Tensor object has a 'requires_grad' parameter. This parameter is a Boolean value which, when True, indicates that the gradient is to be stored, and if False, indicates that the gradient(obj.grad attribute) must be an array of zeros(zero gradient) of the same shape as Tensor array. Setting the 'requires_grad' flag for new Tensors has already been handled.
12. **To implement:**
    - grad_add(): Finding gradients through one addition operation .Ex: Fig1. It returns the gradients of the two operands.
    - grad_matmul(): Finding gradients through one matmul operation. Ex: Fig 2. It returns the gradients of the two operands.
    - backward(): Find the gradients of leaf nodes with respect to the node the backward was called on. Ex: (d = (a+b)@c) (d.backward() the gradients of a,b,c are found and set in their respective 'grad' attributes)
    - **The functions grad_add() and grad_matmul(), backward() take in an argument called 'gradients'. This argument represents the gradients that were computed in the previous stage. You can use these to compute the gradients at the lower stages recursively till the leaf nodes in the graph. Eg: In figure 3 the gradients from 'matmul' can be passed down to 'add'. (Note: This is for you to implement)**
13. Non-Leaf nodes should have zero gradient, that is the gradient(obj.grad attribute) must be an array of zeros of the same shape as the Tensor array.
14. **The gradient of the leaf node has to be set in its 'grad' attribute(leafobject.grad). It is this attribute value you will be evaluated on failure to set the value after computing the correct results will lead to a score of zero.(Evaluation only checks with a precision of 2 so your answer need not be the exact same but has to be accurate to that precision)**.

week4 .py

- ✔ You are provided with the structure of class Tensor.
- ✔ The class Tensor contains one constructor and five methods.
- ✔ Your task is to write code for the grad_add(), grad_matmul() and backward() methods.

1. **You may write your own helper functions if needed**
2. **You can import libraries that come built-in with python 3.7**
3. **You cannot change the skeleton of the code**
4. **Note that the target value is an int**

SampleTest.py

1. This will help you check your code.
2. Passing the cases in this does not ensure full marks, you will need to take care of edge cases
3. Name your code file as YOUR_SRN.py
4. Run the command

   python3 SampleTest.py --SRN YOUR_SRN

   if import error occurs due to any libraries that is mentioned in the skeleton code try:

   python3.7 SampleTest.py --SRN YOUR_SRN

SUBMISSION FORMAT and DEADLINE:

You are required to complete code in week4.py and **rename it to SRN.py**

Failing to submit in the right format will lead to zero marks without a chance for correction

Example: If your SRN is PES1201801819 your file should be PES1201801819.py

- Delete all print statements if used for debugging before submission
- Ensure you run your file against sample test before submitting
- Check for syntax and indentation errors in your file, and make sure that tabs and spaces are used uniformly for indentation (i.e., if tabs are used then only tabs must be used throughout the file, and similarly for spaces)
- All the helper functions should be in the same file (SRN.py), make sure you run the sample test cases as indicated above before submitting.

Soft Deadline (you are expected to complete before this): on or before 2/10/2021 11:59pm

Hard Deadline (final deadline, no extensions post this): on or before 3/10/2021 10pm