MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LAB MANUAL

DEEP LEARNING LAB [CSE 3281]

Sixth Semester BTech in CSE(AI&ML)

(JAN – MAY 2024)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL-576104

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

CERTIFICATE

This is to certify that Ms./Mr. ………………………………………………………………………….

Reg. No.: …………………………………… Section: ………………… Roll No.: ……………………

has satisfactorily completed the **LAB EXERCISES PRESCRIBED FOR DEEP LEARNING LAB (CSE 3281)** of Third Year B.Tech. degree in Computer Science and Engineering (AI & ML) at MIT, Manipal, in the Academic Year 2023– 2024.

Date: ……………………………

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Understand implementation detail of deep learning models.
- Develop familiarity with tools and software frameworks for designing DNNs.

Course Outcomes

At the end of this course, students will be able to

- Understand basic motivation and functioning of the most common type of neural network and its activation functions.
- Design Convolutional Neural Network and perform classification using Convolutional Neural Network.
- Implement some of the important well-known deep neural architectures for Computer Vision/NLP applications.
- Apply different types of auto encoders with dimensionality reduction and regularization.
- Apply deep learning techniques for practical problems.

Evaluation plan

- Internal Assessment Marks: 60M
    - Continuous Evaluation: 20M
        - Continuous evaluation component (for each evaluation): 10 marks
        - The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
    - Mid-term test : 20M
    - Mini-project: 20M [Report 50% + Implementation and Demo 50%]
- End semester assessment: 40

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum.
6. Students must come prepared for the lab in advance.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercise in Lab

- Implement the given exercise individually and not in a group.
- Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
- Solved example
- Lab exercises - to be completed during lab hours
- Additional Exercises - to be completed outside the lab or in the lab to enhance the

- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Lab No 1: Date:

Introduction to tensors

Objectives:

In this lab, student will be able to

1. Setup pytorch environment for deep learning
2. Understand the concept of tensor
3. Manipulate tensors using built-in functions

A summary of the topics that is covered in this session are:

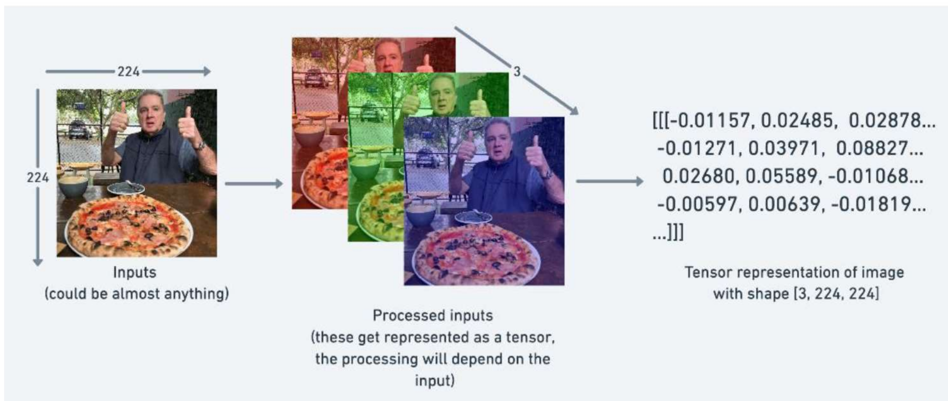
Topic	**Contents**
Introduction to tensors	Tensors are the basic building block of all of machine learning and deep learning.
Creating tensors	Tensors can represent almost any kind of data (images, words, tables of numbers).
Getting information from tensors	If you can put information into a tensor, you'll want to get it out too.
Manipulating tensors	Machine learning algorithms (like neural networks) involve manipulating tensors in many different ways such as adding, multiplying, combining.
Dealing with tensor shapes	One of the most common issues in machine learning is dealing with shape mismatches (trying to mixed wrong shaped tensors with other tensors).
Indexing on tensors	If you've indexed on a Python list or NumPy array, it's very similar with tensors, except they can have far more dimensions.
Mixing PyTorch tensors and NumPy	PyTorch plays with tensors (torch.Tensor), NumPy likes arrays (np.ndarray) sometimes you'll want to mix and match these.
Running tensors on GPU	GPUs (Graphics Processing Units) make your code faster, PyTorch makes it easy to run your code on GPUs.

Sample Exercise:

Use console window to execute the instructions given below:

```
import torch
torch.__version__
```

Introduction to tensors



Creating tensors

```
# Scalar
scalar = torch.tensor(7)
scalar
```

```
# Get the Python number within a tensor (only works with one-element tensors)
scalar.item()
```

```
# Vector
vector = torch.tensor([7, 7])
vector
```

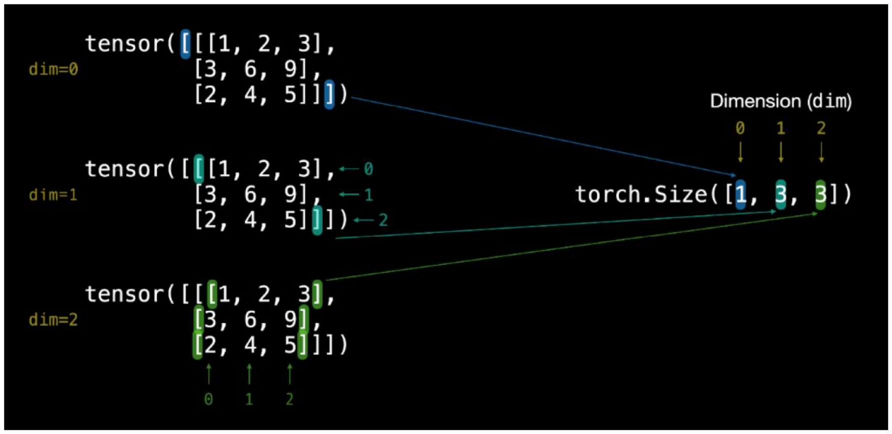
```
# Matrix
MATRIX = torch.tensor([[7, 8],
                       [9, 10]])
MATRIX
```

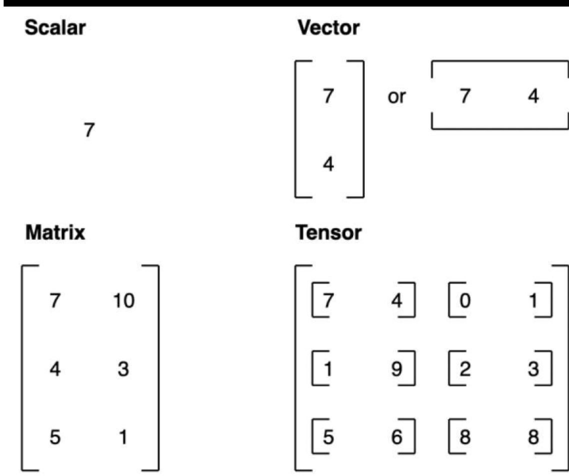
```
MATRIX.shape
```

```
# Tensor
TENSOR = torch.tensor([[[1, 2, 3],
                        [3, 6, 9],
                        [2, 4, 5]]])
TENSOR
```

```
# Check number of dimensions for TENSOR
TENSOR.ndim
```

Visualization of Tensor Dimension:





Random Tensors:

```
# Create a random tensor of size (3, 4)
random_tensor = torch.rand(size=(3, 4))

random_tensor, random_tensor.dtype
```

Output:

```
(tensor([[0.9900, 0.1882, 0.1744, 0.7445],
        [0.9445, 0.7044, 0.7024, 0.7877],
        [0.0218, 0.7861, 0.9037, 0.9690]]),
 torch.float32)
```

The flexibility of torch.rand() is that we can adjust the size to be whatever we want.

For example, say you wanted a random tensor in the common image shape of [224, 224, 3] ([height, width, color_channels]).

```
# Create a random tensor of size (224, 224, 3)
random_image_size_tensor = torch.rand(size=(224, 224, 3))

random_image_size_tensor.shape, random_image_size_tensor.ndim
(torch.Size([224, 224, 3]), 3)
```

Zeros and ones

Sometimes you'll just want to fill tensors with zeros or ones.

This happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).

Let's create a tensor full of zeros with torch.zeros()

Again, the size parameter comes into play.

```
# Create a tensor of all zeros
zeros = torch.zeros(size=(3, 4))
zeros, zeros.dtype
```

Output:

```
(tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]),
 torch.float32)
```

We can do the same to create a tensor of all ones except using torch.ones() instead.

```
# Create a tensor of all ones
ones = torch.ones(size=(3, 4))
ones, ones.dtype
```

Output:

```
(tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]),
 torch.float32)
```

Creating a range and tensors:

Sometimes you might want a range of numbers, such as 1 to 10 or 0 to 100. You can use torch.arange(start, end, step) to do so.

Where:
start = start of range (e.g. 0)
end = end of range (e.g. 10)
step = how many steps in between each value (e.g. 1)

Note: In Python, you can use range() to create a range. However in PyTorch, torch.range() is deprecated and may show an error in the future.

```
# Use torch.arange(), torch.range() is deprecated
zero_to_ten_deprecated = torch.range(0, 10) # Note: this may return an error in the future

# Create a range of values 0 to 10
zero_to_ten = torch.arange(start=0, end=10, step=1)
zero_to_ten

# Can also create a tensor of zeros similar to another tensor
ten_zeros = torch.zeros_like(input=zero_to_ten) # will have same shape
ten_zeros
```

Note:
There are many different tensor datatypes available in PyTorch. Some are specific for CPU and some are better for GPU. Getting to know which is which can take some time. Generally if you see torch.cuda anywhere, the tensor is being used for GPU (since Nvidia GPUs use a computing toolkit called CUDA). The most common type (and generally the default) is torch.float32 or torch.float. This is referred to as "32-bit floating point". But there's also 16-bit floating point (torch.float16 or torch.half) and 64-bit floating point (torch.float64 or torch.double). And to confuse things even more there's also 8-bit, 16-bit, 32-bit and 64-bit integers. The reason for all of these is to do with precision in computing. Precision is the amount of detail used to describe a number. The higher the precision value (8, 16, 32), the more detail and hence data used to express a number. This matters in deep learning and numerical computing because you're making so many operations, the more detail you have to calculate on, the more compute you have to use. So lower precision datatypes

are generally faster to compute on but sacrifice some performance on evaluation metrics like accuracy (faster to compute but less accurate).

Let's see how to create some tensors with specific datatypes. We can do so using the dtype parameter.

```
# Default datatype for tensors is float32

float_32_tensor = torch.tensor([3.0, 6.0, 9.0], dtype=None, device=None,
requires_grad=False)




# dtype=None, defaults to None, which is torch.float32 or whatever datatype is
passed
# device=None, defaults to None, which uses the default tensor type
# requires_grad=False if True, operations performed on the tensor are recorded



float_32_tensor.shape, float_32_tensor.dtype, float_32_tensor.device


float_16_tensor = torch.tensor([3.0, 6.0, 9.0],
                               dtype=torch.float16) # torch.half would also
work

float_16_tensor.dtype

# Create a tensor
some_tensor = torch.rand(3, 4)

# Find out details about it
print(some_tensor)
print(f"Shape of tensor: {some_tensor.shape}")
print(f"Datatype of tensor: {some_tensor.dtype}")
print(f"Device tensor is stored on: {some_tensor.device}") # will default to
CPU

tensor([[0.9270, 0.6217, 0.9093, 0.1493],
        [0.4354, 0.6207, 0.9224, 0.0312],
        [0.3300, 0.0959, 0.6050, 0.7674]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Manipulating tensors (tensor operations)

In deep learning, data (images, text, video, audio, protein structures, etc) gets represented as tensors.

A model learns by investigating those tensors and performing a series of operations on tensors to create a representation of the patterns in the input data.
These operations are often:

- Addition
- Substraction
- Multiplication (element-wise)
- Division
- Matrix multiplication

Basic operations
Let's start with a few of the fundamental operations, addition (+), subtraction (-), mutliplication (*).

They work just as you think they would.

```
# Create a tensor of values and add a number to it
tensor = torch.tensor([1, 2, 3])
tensor + 10
tensor([11, 12, 13])

# Multiply it by 10
tensor * 10
tensor([10, 20, 30])

Notice how the tensor values above didn't end up being tensor([110, 120,
130]), this is because the values inside the tensor don't change unless
they're reassigned.

# Tensors don't change unless reassigned
tensor
tensor([1, 2, 3])
Let's subtract a number and this time we'll reassign the tensor variable.
# Subtract and reassign
tensor = tensor - 10
tensor
tensor([-9, -8, -7])
# Add and reassign
tensor = tensor + 10
tensor
tensor([1, 2, 3])

PyTorch also has a bunch of built-in functions like torch.mul() (short for
multiplcation) and torch.add() to perform basic operations.
# Can also use torch functions
torch.multiply(tensor, 10)
tensor([10, 20, 30])
# Original tensor is still unchanged
tensor
tensor([1, 2, 3])
```

However, it's more common to use the operator symbols like * instead of torch.mul()

```
# Element-wise multiplication (each element multiplies its equivalent, index 0->0, 1->1, 2->2)
print(tensor, "*", tensor)
print("Equals:", tensor * tensor)
tensor([1, 2, 3]) * tensor([1, 2, 3])
Equals: tensor([1, 4, 9])
```

Matrix multiplication:

One of the most common operations in machine learning and deep learning algorithms (like neural networks) is matrix multiplication. PyTorch implements matrix multiplication functionality in the torch.matmul() method.

The main two rules for matrix multiplication to remember are:

1. The **inner dimensions** must match:
   - (3, 2) @ (3, 2) won't work
   - (2, 3) @ (3, 2) will work
   - (3, 2) @ (2, 3) will work
2. The resulting matrix has the shape of the **outer dimensions**:
   - (2, 3) @ (3, 2) -> (2, 2)
   - (3, 2) @ (2, 3) -> (3, 3)

Let's create a tensor and perform element-wise multiplication and matrix multiplication on it.

```
import torch
tensor = torch.tensor([1, 2, 3])
tensor.shape
torch.Size([3])
```

The difference between element-wise multiplication and matrix multiplication is the addition of values.

For our tensor variable with values [1, 2, 3]:

Operation	Calculation	Code
Element-wise multiplication	[1*1, 2*2, 3*3] = [1, 4, 9]	tensor * tensor
Matrix multiplication	[1*1 + 2*2 + 3*3] = [14]	tensor.matmul(tensor)

```
# Element-wise matrix multiplication
tensor * tensor
tensor([1, 4, 9])
# Matrix multiplication
torch.matmul(tensor, tensor)
tensor(14)
# Can also use the "@" symbol for matrix multiplication, though not recommended
tensor @ tensor
tensor(14)
```

You can do matrix multiplication by hand but it's not recommended.

The in-built torch.matmul() method is faster.

```
%%time
# Matrix multiplication by hand
# (avoid doing operations with for loops at all cost, they are computationally
expensive)
value = 0
for i in range(len(tensor)):
  value += tensor[i] * tensor[i]
value
CPU times: user 178 µs, sys: 62 µs, total: 240 µs
Wall time: 248 µs

tensor(14)
%%time
torch.matmul(tensor, tensor)
CPU times: user 272 µs, sys: 94 µs, total: 366 µs
Wall time: 295 µs

tensor(14)
```

Getting PyTorch to run on the GPU

You can test if PyTorch has access to a GPU using `torch.cuda.is_available()`.

```
# Check for GPU
import torch
torch.cuda.is_available()
False
```

Let's create a device variable to store what kind of device is available.

```
# Set device type
device = "cuda" if torch.cuda.is_available() else "cpu"
device
'cpu'
```

```
# Count number of devices
torch.cuda.device_count()
```

Putting tensors (and models) on the GPU

You can put tensors (and models, we'll see this later) on a specific device by calling to(device) on them. Where device is the target device you'd like the tensor (or model) to go to.

Why do this?

GPUs offer far faster numerical computing than CPUs do and if a GPU isn't available, because of our device agnostic code (see above), it'll run on the CPU.

Note: Putting a tensor on GPU using to(device) (e.g. some_tensor.to(device)) returns a copy of that tensor, e.g. the same tensor will be on CPU and GPU. To overwrite tensors, reassign them:

`some_tensor = some_tensor.to(device)`

Let's try creating a tensor and putting it on the GPU (if it's available).

```
# Create tensor (default on CPU)
tensor = torch.tensor([1, 2, 3])

# Tensor not on GPU
print(tensor, tensor.device)

# Move tensor to GPU (if available)
tensor_on_gpu = tensor.to(device)
tensor_on_gpu
```

```
tensor([1, 2, 3]) cpu
tensor([1, 2, 3], device='cuda:0')
```

Moving tensors back to the CPU

What if we wanted to move the tensor back to CPU?
For example, you'll want to do this if you want to interact with your tensors with NumPy (NumPy does not leverage the GPU).
Let's try using the torch.Tensor.numpy() method on our tensor_on_gpu.

```
# If tensor is on GPU, can't transform it to NumPy (this will error)
tensor_on_gpu.numpy()
```

Instead, to get a tensor back to CPU and usable with NumPy we can use Tensor.cpu(). This copies the tensor to CPU memory so it's usable with CPUs.

```
# Instead, copy the tensor back to cpu
tensor_back_on_cpu = tensor_on_gpu.cpu().numpy()
tensor_back_on_cpu
```

Lab Exercise:

1. Illustrate the functions for Reshaping, viewing, stacking, squeezing and unsqueezing of tensors
2. Illustrate the use of torch.permute().
3. Illustrate indexing in tensors
4. Show how numpy arrays are converted to tensors and back again to numpy arrays
5. Create a random tensor with shape (7, 7).
6. Perform a matrix multiplication on the tensor from 2 with another random tensor with shape (1, 7) (hint: you may have to transpose the second tensor).
7. Create two random tensors of shape (2, 3) and send them both to the GPU (you'll need access to a GPU for this).
8. Perform a matrix multiplication on the tensors you created in 6 (again, you may have to adjust the shapes of one of the tensors).
9. Find the maximum and minimum values of the output of 7.
10. Find the maximum and minimum index values of the output of 7.

11. Make a random tensor with shape (1, 1, 1, 10) and then create a new tensor with all the 1 dimensions removed to be left with a tensor of shape (10). Set the seed to 7 when you create it and print out the first tensor and it's shape as well as the second tensor and it's shape.

Lab No 2: Date:

Introduction to tensors

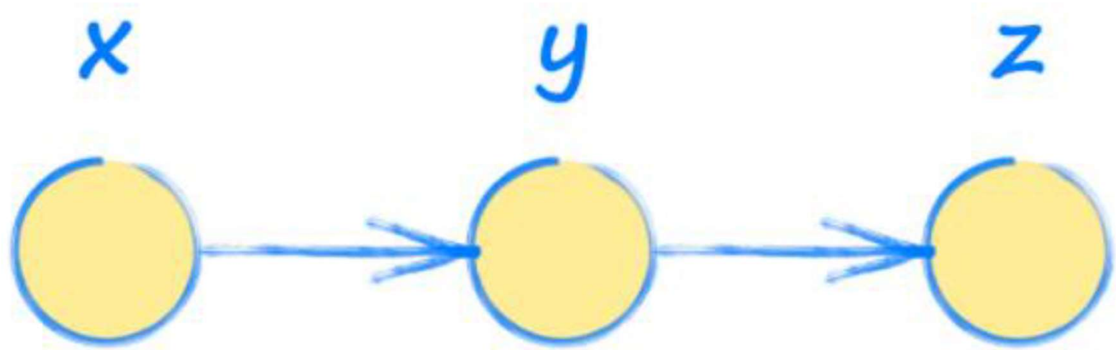
Objectives:

In this lab, student will be able to

- Calculate derivatives in PyTorch and perform auto differentiation on tensors.
- Calculate partial derivatives in PyTorch and implement the derivative of functions with respect to multiple values.
- Create a computation graph that involves different nodes and leaves to calculate the gradients using the chain rule to visualize the prediction (forward path) and parameter learning (backward path) in a step-wise fashion.
- To use computation graphs to make complicated mathematical concepts of learning algorithms more intuitive.

A summary of the topics that is covered in this session are:

To better understand neural networks, it is important to practice with computation graphs. These graphs are essentially a simplified version of neural networks with a sequence of operations used to see how the output of a system is affected by the input.



In other words, input x is used to find y, which is then used to find the output z.

PyTorch allows to automatically obtain the gradients of a tensor with respect to a defined function. When creating the tensor, we have to indicate that it requires the gradient computation using the flag requires_grad

Sample Program:

```
x = torch.rand(3,requires_grad=True)

print(x)

tensor([0.9207, 0.2854, 0.1424], requires_grad=True)
```

Notice that now the Tensor shows the flag requires_grad as True. We can also activate such a flag in a Tensor already created as follows:

x = torch.tensor([1.0,2.0,3.0])

x.requires_grad_(True)

print(x)

tensor([1., 2., 3.], requires_grad=True)

Problem 1:

Consider that y and z are calculated as follows:

$$y = x^2$$

$$z = 2y + 3$$

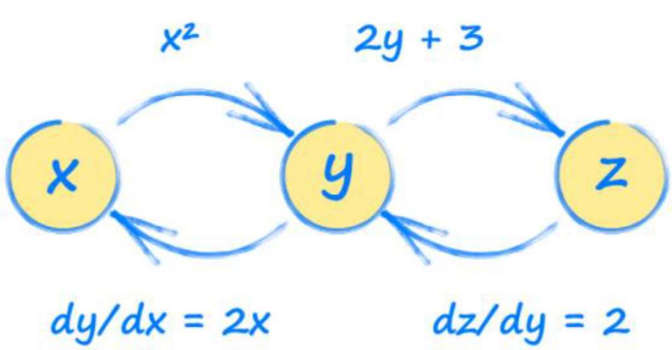
We are interested in how output z changes with input x

$$\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx}$$

$$\frac{dz}{dx} = 2.2x$$

$$\frac{dz}{dx} = 4x$$

For input x=3.5, will make z = 14



This picture is called a computation graph. Using this graph, we can see how each tensor will be affected by a change in any other tensor. These relationships are gradients and are used to update a neural network during training

```
import torch

# set up simple graph relating x, y and z

x = torch.tensor(3.5, requires_grad=True)

y = x*x

z = 2*y + 3

print("x: ", x)

print("y = x*x: ", y)

print("z= 2*y + 3: ", z)

# work out gradients

z.backward()

print("Working out gradients dz/dx")

# what is gradient at x = 3.5

print("Gradient at x = 3.5: ", x.grad)
```

```
x:  tensor(3.5000, requires_grad=True)

y = x*x:  tensor(12.2500, grad_fn=<MulBackward0>)

z= 2*y + 3:  tensor(27.5000, grad_fn=<AddBackward0>)

Working out gradients dz/dx

Gradient at x = 3.5:  tensor(14.)
```

Problem 2:

Consider the function $f(x)=(x-2)^2$. Compute $d/dx\ f(x)$ and then compute $f'(1)$. Write code to check analytical gradient.

```
def f(x):

   return (x-2)**2

def fp(x):

   return 2*(x-2)
```

```
x = torch.tensor([1.0], requires_grad=True)
y = f(x)
y.backward()

print('Analytical f\'(x):', fp(x))
print('PyTorch\'s f\'(x):', x.grad)
```

Analytical f'(x): tensor([-2.], grad_fn=<MulBackward0>)

PyTorch's f'(x): tensor([-2.])

Problem 3:

Define a function $y = x^2 + 5$. The function y will not only carry the result of evaluating x, but also the gradient function $\frac{\partial y}{\partial x}$ called grad_fn in the new tensor y . Compare the result with analytical gradient.

```
x = torch.tensor([2.0])
x.requires_grad_(True)  #indicate we will need the gradients with respect to this variable
y = x**2 + 5
print(y)
```

tensor([9.], grad_fn=<AddBackward0>)

To evaluate the partial derivative $\frac{\partial y}{\partial x}$, we use the .backward() function and the result of the gradient evaluation is stored in x.grad

```
y.backward()  #dy/dx
print('PyTorch gradient:', x.grad)

#Let us compare with the analytical gradient of y = x**2+5
```

```
with torch.no_grad():   #this is to only use the tensor value without its gradient information
    dy_dx = 2*x  #analytical gradient
print('Analytical gradient:',dy_dx)
```

PyTorch gradient: tensor([4.])

Analytical gradient: tensor([4.])

Problem 4:

Write a function to compute the gradient of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

Write $\sigma(x)$ as a composition of several elementary functions, as $\sigma(x) = s\left(c\left(b\left(a(x)\right)\right)\right)$

where: $a(x) = -x$

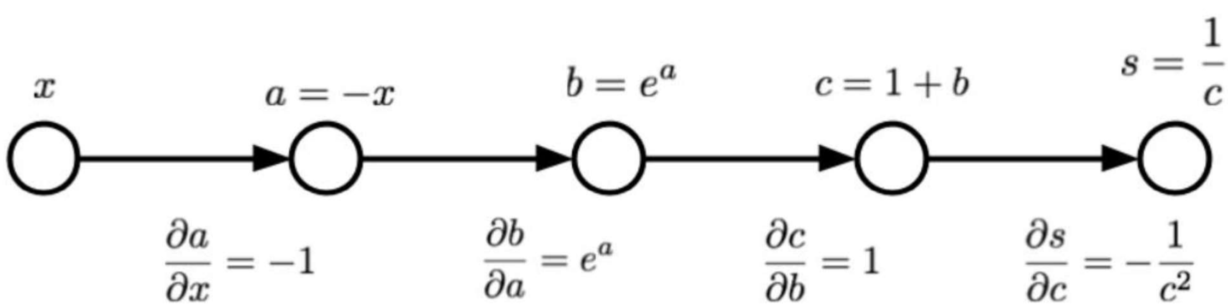
$b(a) = e^a$

$c(b) = 1 + b$

$s(c) = \frac{1}{c}$

It contains several intermediate variables, each of which are basic expressions for which we can easily compute the local gradients.

The computation graph for this expression is shown in the figure below



The input to this function is x, and the output is represented by node s. Compute the gradient of s with respect to x, $\frac{\partial s}{\partial x}$. In order to make use of our intermediate computations, we can use the chain rule as follows:

$$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

```
def grad_sigmoid_manual(x):
    """Implements the gradient of the logistic sigmoid function
    #sigma(x) = 1 / (1 + e^{-x})
    """
    # Forward pass, keeping track of intermediate values for use in the
    # backward pass
    a = -x         # -x in denominator
    b = np.exp(a)  # e^{-x} in denominator
    c = 1 + b      # 1 + e^{-x} in denominator
    s = 1.0 / c    # Final result, 1.0 / (1 + e^{-x})

    # Backward pass
    dsdc = (-1.0 / (c**2))
    dsdb = dsdc * 1
    dsda = dsdb * np.exp(a)
    dsdx = dsda * (-1)

    return dsdx



def sigmoid(x):
    y = 1.0 / (1.0 + torch.exp(-x))
    return y


input_x = 2.0
```

```
x = torch.tensor(input_x).requires_grad_(True)
y = sigmoid(x)
y.backward()

# Compare the results of manual and automatic gradient functions:
print('autograd:', x.grad.item())
print('manual:', grad_sigmoid_manual(input_x))
```

autograd: 0.10499356687068939

manual: 0.1049935854035065

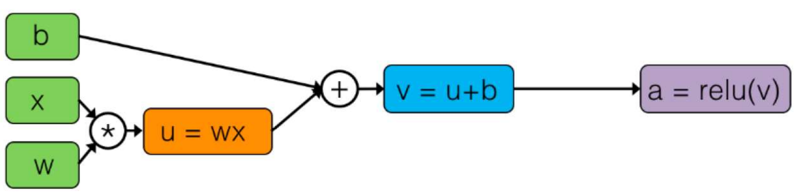
Exercise Questions:

1. Draw Computation Graph and work out the gradient dz/da by following the path back from z to a and compare the result with the analytical gradient.

   x = 2*a + 3*b

   y = 5*a*a + 3*b*b*b

   z = 2*x + 3*y
2. For the following Computation Graph, work out the gradient da/dw by following the path back from a to w and compare the result with the analytical gradient.



3. Repeat the Problem 2 using Sigmoid function
4. Verify that the gradients provided by PyTorch match with the analytical gradients of the function f= $\exp(-x^2-2x-\sin(x))$ w.r.t x
5. Compute gradient for the function $y=8x^4+ 3x^3 +7x^2+6x+3$ and verify the gradients provided by PyTorch with the analytical gradients. A snapshot of the Python code is provided below.

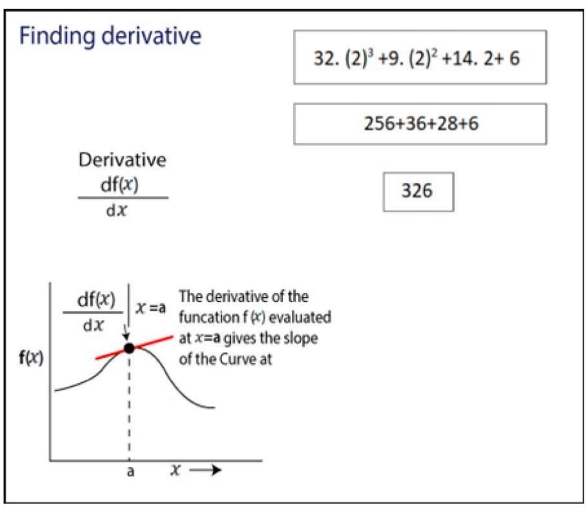
$$\boxed{8x^4 + 3x^3 + 7x^2 + 6x + 3}$$

$$\frac{d}{dx}[8x^4 + 3x^3 + 7x^2 + 6x + 1]$$

$$= 8.\frac{d}{dx}[x^4] + 3.\frac{d}{dx}[x^3] + 7.\frac{d}{dx}[x^2] + 6.\frac{d}{dx}[x] + \frac{d}{dx}[1]$$

$$= 8.4x^3 + 3.3x^2 + 7.2x + 6.1 + 0$$

$$\boxed{= 32x^3 + 9x^2 + 14x + 6}$$

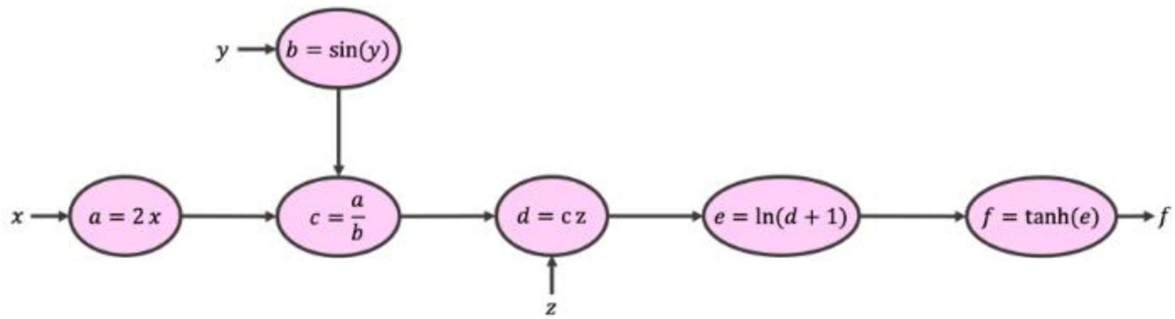



```
import torch
x=torch.tensor(2.0, requires_grad=True)
y=8*x**4+3*x**3+7*x**2+6*x+3
y.backward()
x.grad
```

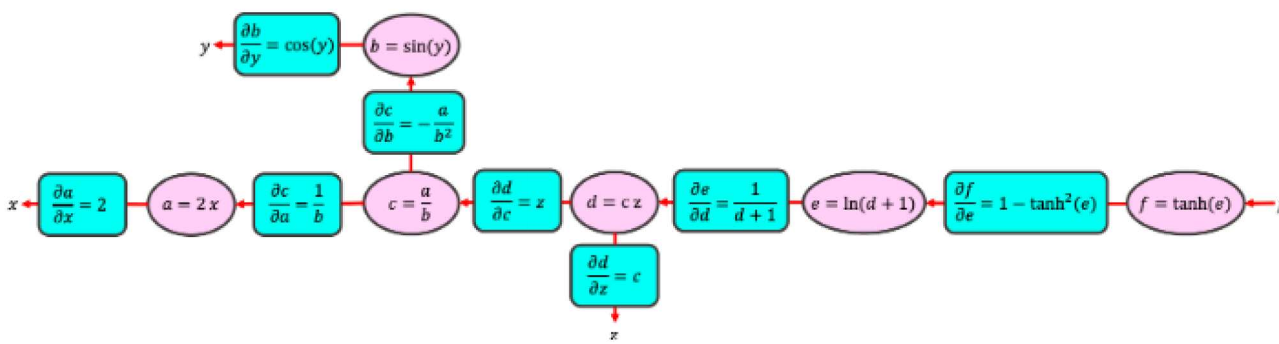
tensor(326.)

6. For the following function, computation graph is provided below.

$$f(x, y, z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$



Calculate the intermediate variables a,b,c,d, and e in the forward pass. Starting from f, calculate the gradient of each expression in the backward pass manually. Calculate ∂f/∂y using the computational graph and chain rule. Use the chain rule to calculate gradient and compare with analytical gradient.



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial e}\,\frac{\partial e}{\partial d}\,\frac{\partial d}{\partial c}\,\frac{\partial c}{\partial a}\,\frac{\partial a}{\partial x} = \left(1 - \tanh^2(e)\right) \cdot \frac{1}{d+1} \cdot z \cdot \frac{1}{b} \cdot 2$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial e}\,\frac{\partial e}{\partial d}\,\frac{\partial d}{\partial c}\,\frac{\partial c}{\partial b}\,\frac{\partial b}{\partial y} = \left(1 - \tanh^2(e)\right) \cdot \frac{1}{d+1} \cdot z \cdot \frac{-a}{b^2} \cdot \cos(y)$$

References:

1. Eli Stevens, Luca Antiga, and Thomas Viehmann, Deep Learning with PyTorch, Manning, 2020
2. Goodfellow, Ian, et al. Deep learning. Vol. 1. No. 2. Cambridge: MIT press, 2016.