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:

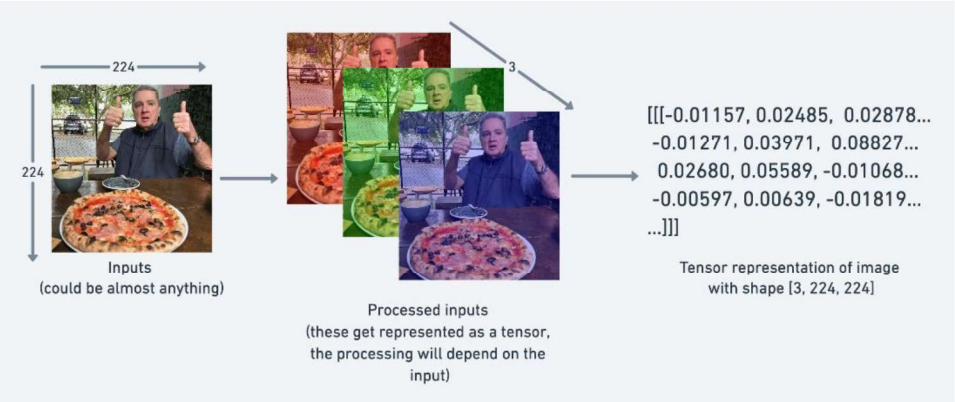
Topic	Contents
Introduction to tensors	Tensors are the basic building block of all of machine learning and deep learning.
Creating tensors	Tensors can represent almost any kind of data (images, words, tables of numbers).
Getting information from tensors	If you can put information into a tensor, you'll 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
```

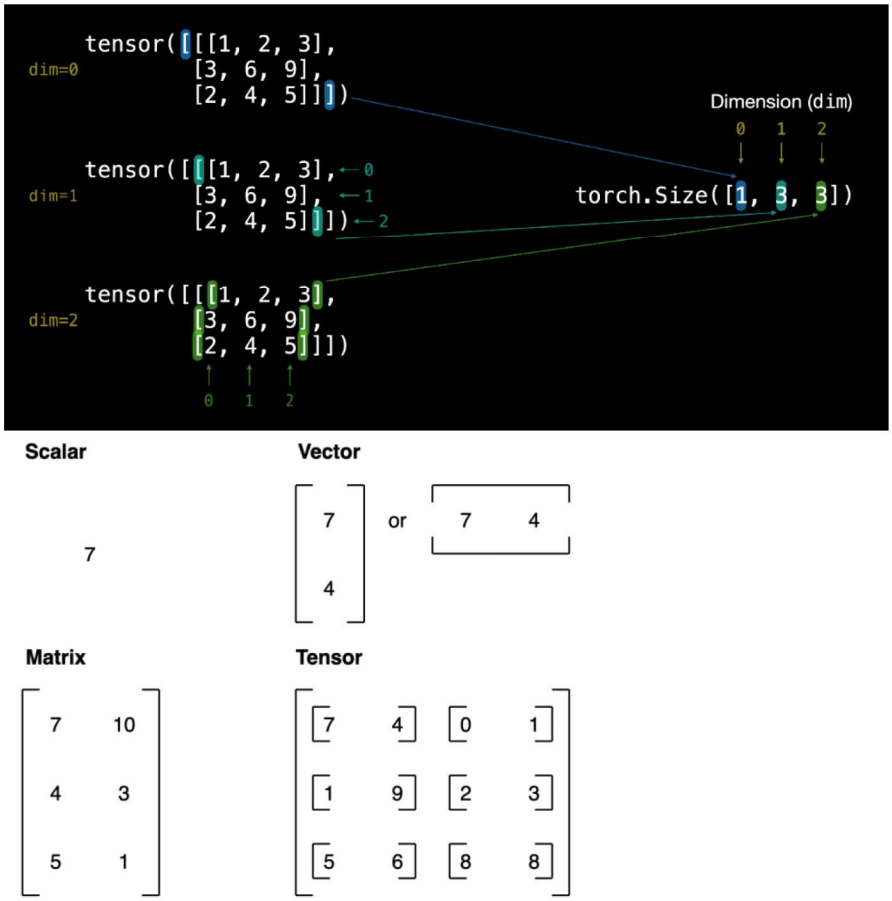
```
# Matrix
MATRIX = torch.tensor([[7, 8],
                       [9, 10]])
MATRIX
```

```
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:

Computation Graphs

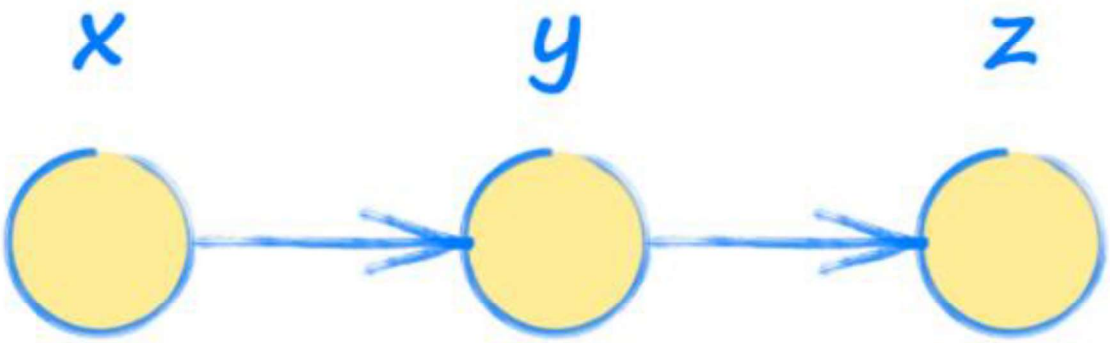
Objectives:

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:

$$y = x^2$$

$$z = 2y + 3$$

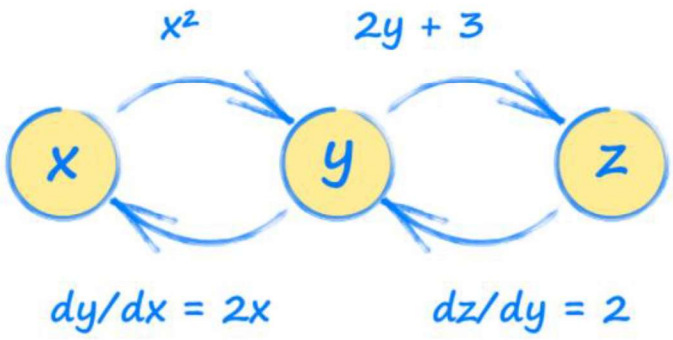
We are interested in how output z changes with input x

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

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

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

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 $\sigma(x) = s\left(c\left(b(a(x))\right)\right)$

where: a(x) = -x

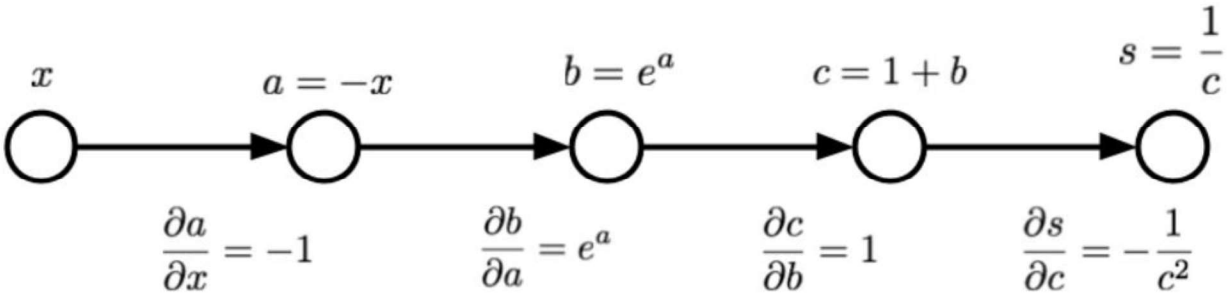
$b(a) = e^a$

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:

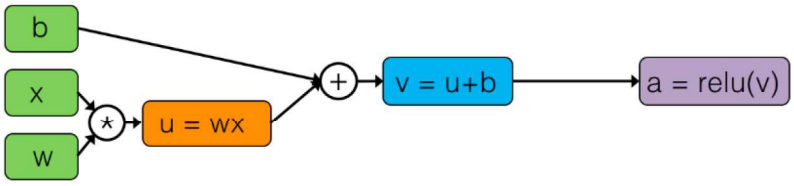
1. Draw Computation Graph and work out the gradient dz/da by following the path back from z to a and compare the result with the analytical gradient.

   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.

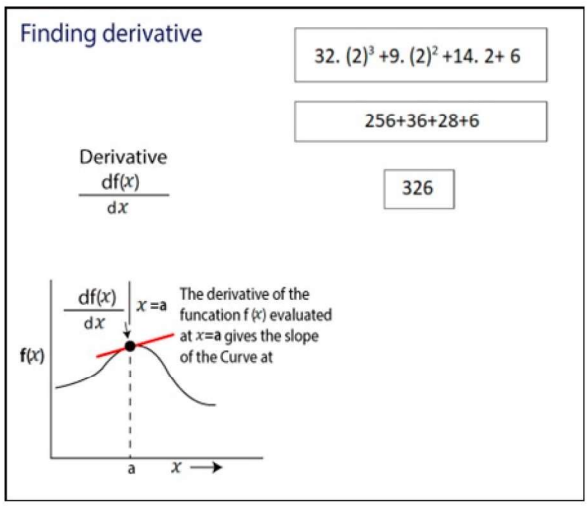
$$8x^4 + 3x^3 + 7x^2 + 6x + 3$$

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

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

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

$$= 32x^3 + 9x^2 + 14x + 6$$

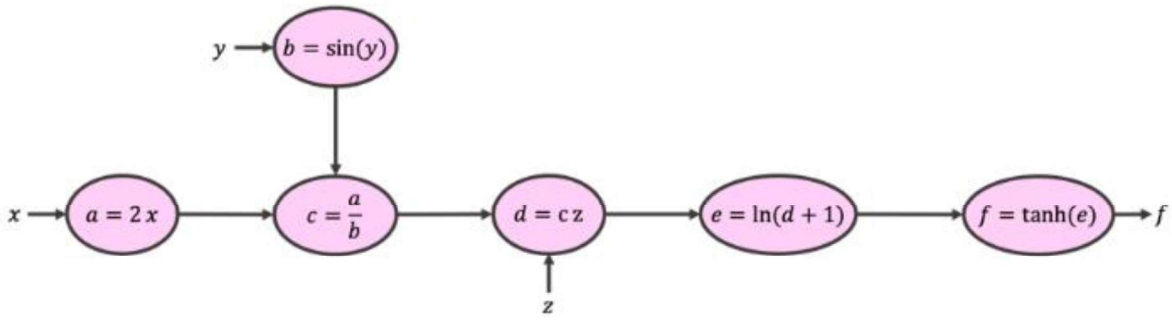



```
import torch
x=torch.tensor(2.0, requires_grad=True)
y=8*x**4+3*x**3+7*x**2+6*x+3
y.backward()
x.grad
```

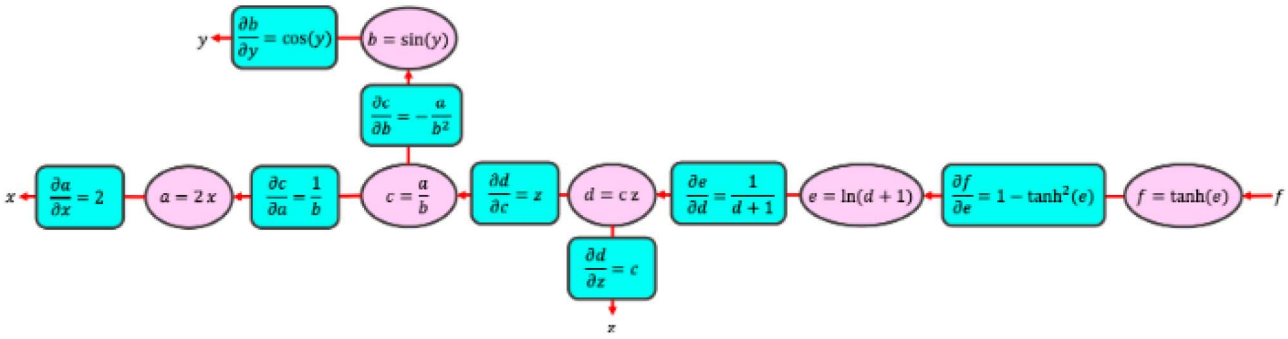
tensor(326.)

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.



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

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

Lab No 3: Date:

Deep Learning Library in PyTorch

Objectives:

In this lab, student will be able to

- To build neural networks from scratch, starting off with a simple linear regression model.
- Develop PyTorch deep learning models for predictive modeling tasks such as linear regression and classification.
- Using the PyTorch API for deep learning model development tasks such as linear regression
- To explore multiple ways of implementing linear regression and logistic regression using PyTorch

Linear regression

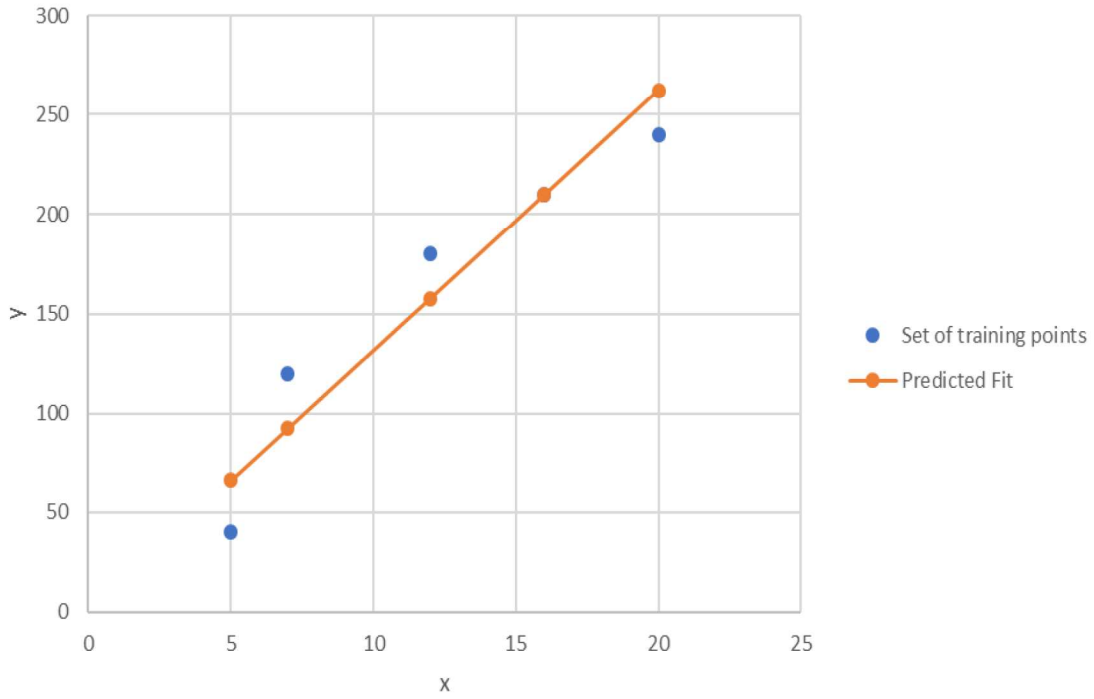
Linear regression is a linear model, a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x).

- When there is a single input variable (x), the method is referred to as simple linear regression.
- When there are multiple input variables, multiple linear regression.

Allows us to understand relationship between two continuous variables
Example
x: independent variable
weight
y: dependent variable
height
$y=wx+b$

Aim of Linear Regression: Minimize the distance between the points and the line ($y=\alpha x+\beta$)
Adjusting Coefficient: w and Bias/intercept: b
Learnable parameters: w, b

X	Y
5	40
7	120
12	180
16	210
20	240



In order to train a linear regression model, we need to define a cost function and an optimizer.

The cost function is used to measure how well our model fits the data, while the optimizer decides which direction to move in order to improve this fit.

Cost function: **MSE Loss - Mean Squared Error**

- $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
  - $\hat{y}$: prediction
  - y: true value

Optimizer: Gradient Descent

- Simplified equation
  - $\theta = \theta - \eta \cdot \nabla_\theta$
    - θ: parameters (our variables)
    - η: learning rate (how fast we want to learn)
    - ∇_θ: parameters' gradients
- Even simplier equation
  - `parameters = parameters - learning_rate * parameters_gradients`

Problem 1: Building a Linear Regression Model

For the following training data, build a regression model. Assume w and b is initialized with 1 and learning parameter is set to 0.001.

x = torch.tensor([5.0, 7.0, 12.0, 16.0, 20.0])
y = torch.tensor([40.0, 120.0, 180.0, 210.0, 240.0])

```
import torch
from matplotlib import pyplot as plt

# Create the tensors x and y. They are the training
# examples in the dataset for the linear regression
x = torch.tensor(
    [12.4, 14.3, 14.5, 14.9, 16.1, 16.9, 16.5, 15.4, 17.0, 17.9, 18.8, 20.3, 22.4, 19.4, 15.5, 16.7, 17.3, 18.4, 19.2,
     17.4, 19.5, 19.7, 21.2])
y = torch.tensor(
    [11.2, 12.5, 12.7, 13.1, 14.1, 14.8, 14.4, 13.4, 14.9, 15.6, 16.4, 17.7, 19.6, 16.9, 14.0, 14.6, 15.1, 16.1, 16.8,
     15.2, 17.0, 17.2, 18.6])

# The parameters to be learnt w, and b in the
# prediction y_p = wx +b
b = torch.rand([1], requires_grad=True)
w = torch.rand([1], requires_grad=True)
print("The parameters are {}, and {}".format(w, b))


# The learning rate is set to alpha = 0.001
learning_rate = torch.tensor(0.001)

#The list of loss values for the plotting purpose
loss_list = []
```

```
# Run the training loop for N epochs
for epochs in range(100):
    #Compute the average loss for the training samples
    loss = 0.0
    #Accumulate the loss for all the samples
    for j in range(len(x)):
        a = w * x[j]
        y_p = a + b
        loss += ( y_p-y[j]) ** 2
    #Find the average loss
    loss = loss / len(x)
    #Add the loss to a list for the plotting purpose
    loss_list.append(loss.item())
    #Compute the gradients using backward
    # dL/dw and dL/db
    loss.backward()
    # Without modifying the gradient in this block
    # perform the operation
    with torch.no_grad():
        # Update the weight based on gradient descent
        # equivalently one may write w1.copy_(w1 - learning_rate * w1.grad)
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad
    #reset the gradients for next epoch
    w.grad.zero_()
    b.grad.zero_()
    #w.grad = None
    #b.grad = None
    # prev_loss = loss
    #Display the parameters and loss
    print("The parameters are w={},  b={}, and loss={}".format(w, b, loss.item()))
#Display the plot
plt.plot(loss_list)
plt.show()
```

Exercise Questions:

1. For the following training data, build a linear regression model. Assume w and b are initialized with 1 and learning parameter is set to 0.001.

x = torch.tensor([12.4, 14.3, 14.5, 14.9, 16.1, 16.9, 16.5, 15.4, 17.0, 17.9, 18.8, 20.3, 22.4, 19.4, 15.5, 16.7, 17.3, 18.4, 19.2, 17.4, 19.5, 19.7, 21.2])

y = torch.tensor([11.2, 12.5, 12.7, 13.1, 14.1, 14.8, 14.4, 13.4, 14.9, 15.6, 16.4, 17.7, 19.6, 16.9, 14.0, 14.6, 15.1, 16.1, 16.8, 15.2, 17.0, 17.2, 18.6])
Assume learning rate =0.001. Plot the graph of epoch in x axis and loss in y axis.

2. Find the value of w.grad, b.grad using analytical solution for the given linear regression problem. Initial value of w = b =1. Learning parameter is set to 0.001. Implement the same and verify the values of w.grad , b.grad and updated parameter values for two epochs. Consider the difference between predicted and target values of y is defined as (yp-y).

x	y
2	20
4	40

3. Revise the linear regression model by defining a user defined class titled RegressionModel with two parameters w and b as its member variables. Define a constructor to initialize w and b with value 1. Define four member functions namely forward(x) to implement wx+b, update() to update w and b values, reset_grad() to reset parameters to zero, criterion(y, yp) to implement MSE Loss given the predicted y value yp and the target label y. Define an object of this class named *model* and invoke all the methods. Plot the graph of epoch vs loss by varying epoch to 100 iterations.

x = torch.tensor([5.0, 7.0, 12.0, 16.0, 20.0])
y = torch.tensor([40.0, 120.0, 180.0, 210.0, 240.0])

learning_rate = torch.tensor(0.001)

```python
class RegressionModel:
    def __init__(self):
        self.w = torch.rand([1], requires_grad=True)
        self.b = torch.rand([1], requires_grad=True)
    def forward(self, x):
        return self.w * x + self.b
    def update(self):
        # Update the weight based on gradient descent
        # equivalently one may write w1.copy_(w1 - learning_rate * w1.grad)
        self.w -= learning_rate * self.w.grad
        self.b -= learning_rate * self.b.grad

    def reset_grad(self):
        self.w.grad.zero_()
        self.b.grad.zero_()


def criterion(yj, y_p):
    return (yj - y_p)**2
```

```
model = RegressionModel()

#The list of loss values for the plotting purpose
loss_list = []

# Run the training loop for N epochs
for epochs in range(100):
    loss = 0.0
    #Accumulate the loss for all the samples
    for j in range(len(x)):
        y_p = model.forward(x[j])
        loss +=criterion(y[j], y_p)

    #Find the average loss
    loss = loss / len(x)
    #Add the loss to a list for the plotting purpose
    loss_list.append(loss.item())
```

```
    #Compute the gradients using backward dl/dw and dl/db
    loss.backward()

    # Without modifying the gradient in this block
    # perform the operation
    with torch.no_grad():
        model.update()
    #reset the gradients for next epoch
    model.reset_grad()
    #w.grad = None
    #b.grad = None

    # prev_loss = loss
    #Display the parameters and loss
    print("The parameters are w={},  b={}, and loss={}".format(model.w, model.b, loss.item()))

#Display the plot
plt.plot(loss_list)
plt.show()
```

4. Convert your program written in Qn 3 to extend nn.module in your model. Also override the necessary methods to fit the regression line. Illustrate the use of Dataset and DataLoader from torch.utils.data in your implementation. Use the SGD Optimizer torch.optim.SGD()

5. Use PyTorch's nn.Linear() in your implementation to perform linear regression for the data provided in Qn. 1. Also plot the graph.

6. Implement multiple linear regression for the data provided below

Subject	X1	X2	Y
1	3	8	-3.7
2	4	5	3.5
3	5	7	2.5
4	6	3	11.5
5	2	1	5.7

Verify your answer for the data point X1=3, X2=2.

7. Implement logistic regression

   x = [1, 5, 10, 10, 25, 50, 70, 75, 100,]
   y = [0, 0, 0, 0, 0, 1, 1, 1, 1]

Additional Question:

1. Find the value of w.grad, b.grad using analytical solution for the given linear regression problem. Initial value of w = b =1. Learning parameter is set to 0.001. Implement the same and verify the values of w.grad , b.grad and updated parameter values for two epochs.

Consider the difference between predicted and target values of y is defined as (y-yp).

x	y
2	20
4	40

Lab No 4: Date:

Feedforward Neural Network

Objectives:

In this lab, student will be able to

- To build single and multi-layered neural networks
- Develop PyTorch deep learning models for tasks such as XOR problem and MNIST digit classification.

A feedforward neural network is a type of artificial neural network in which nodes' connections do not form a loop. The feed forward model is a basic type of neural network because the input is only processed in one direction.

Often referred to as a multi-layered network of neurons, feedforward neural networks are so named because all information flows in a forward manner only.

The data enters the input nodes, travels through the hidden layers, and eventually exits the output nodes. The network is devoid of links that would allow the information exiting the output node to be sent back into the network.

A Feedforward Neural Network Layers

The following are the components of a feedforward neural network:

Layer of input

It contains the neurons that receive input. The data is subsequently passed on to the next tier. The input layer's total number of neurons is equal to the number of variables in the dataset.

Hidden layer

This is the intermediate layer, which is concealed between the input and output layers. This layer has a large number of neurons that perform alterations on the inputs. They then communicate with the output layer. These layers use activation functions, such as ReLU or

sigmoid, to introduce non-linearity into the network, allowing it to learn and model more complex relationships between the inputs and outputs.

Output layer

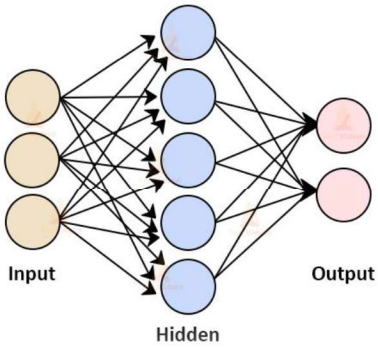
It is the last layer and is depending on the model's construction. Additionally, the output layer is the expected feature or the desired outcome. The output layer generates the final output. Depending on the type of problem, the number of neurons in the output layer may vary. For example, in a binary classification problem, it would only have one neuron. In contrast, a multi-class classification problem would have as many neurons as the number of classes.

Neurons weights

Weights are used to describe the strength of a connection between neurons. The range of a weight's value is from 0 to 1.

When there is a non-linear and complex relationship between X and Y, nevertheless, a Linear Regression method may struggle to predict Y. To approximate that relationship, we may need a curve or a multi-dimensional curve in this scenario.

Neural Network Architecture



A weight is being applied to each input to an artificial neuron. First, the inputs are multiplied by their weights, and then a bias is applied to the outcome. This is called the weighted sum. After that, the weighted sum is processed via an activation function, as a non-linear function.

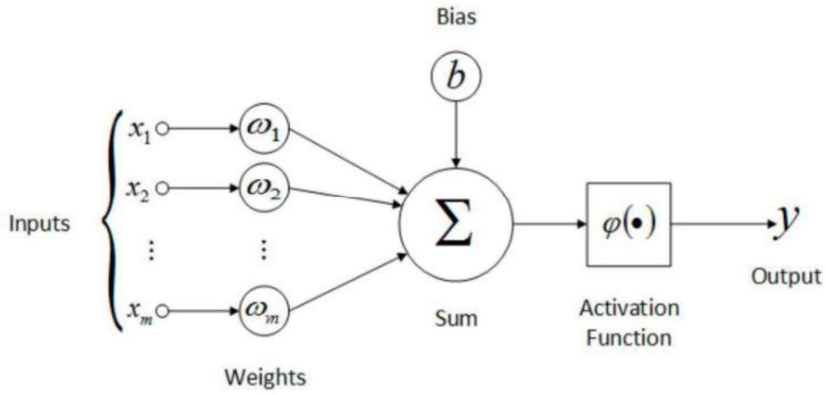
The first layer is the input layer, which appears to have six neurons but is only the data that is sent into the neural network. The output layer is the final layer. The dataset and the type of challenge determine the number of neurons in the final layer and the first layer. Trial and error will be used to determine the number of neurons in the hidden layers and the number of hidden layers.

All of the inputs from the previous layer will be connected to the first neuron from the first hidden layer. The second neuron in the first hidden layer will be connected to all of the preceding layer's inputs, and so forth for all of the first hidden layer's neurons. The outputs of the previously hidden layer are regarded inputs for neurons in the second hidden layer, and each of these neurons is coupled to all of the preceding neurons.

The number of neurons and layers in the hidden layers is one of the hyperparameters that can be adjusted during the design and training of the network. Generally speaking, the more neurons and layers there are, the more complex and abstract features the network can learn. However, this also increases the risk of overfitting and requires more computational power to train the network.

Single Layer feed forward network:

In its most basic form, a Feed-Forward Neural Network is a single layer perceptron. A sequence of inputs enter the layer and are multiplied by the weights in this model. The weighted input values are then summed together to form a total. If the sum of the values is more than a predetermined threshold, which is normally set at zero, the output value is usually 1, and if the sum is less than the threshold, the output value is usually -1. The single-layer perceptron is a popular feed-forward neural network model that is frequently used for classification.



The neural network can compare the outputs of its nodes with the desired values using a property known as the delta rule, allowing the network to alter its weights through training to create more accurate output values. This training and learning procedure results in gradient descent.

Importance of the Non-Linearity

When two or more linear objects, such as a line, plane, or hyperplane, are combined, the outcome is also a linear object: line, plane, or hyperplane. No matter how many of these linear things we add, we'll still end up with a linear object.

However, this is not the case when adding non-linear objects. When two separate curves are combined, the result is likely to be a more complex curve.

We're introducing non-linearity at every layer using these activation functions, in addition to just adding non-linear objects or hyper-curves like hyperplanes. In other words, we're applying a nonlinear function on an already nonlinear object.

What if activation functions were not used in neural networks?

Suppose if neural networks didn't have an activation function, they'd just be a huge linear unit that a single linear regression model could easily replace.

a = m*x + d

Z= k*a + t => k*(m*x+d) + t => k*m*x + k*d + t => (k*m)*x + (k*c+t)

Activation Function

An activation function is a mathematical function applied to a neuron's output in a neural network feedforward. It introduces non-linearity into the network, allowing it to learn and model more complex relationships between the inputs and outputs. Without the activation function, a neural network would be linear, less powerful, and less expressive than a non-linear model.

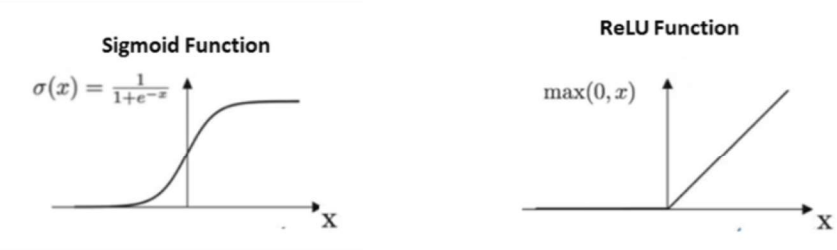
There are many different activation functions that we can use in a neural networks feedforward; some of the most common ones include the following:

Sigmoid:

The sigmoid activation function maps any input value to a value between 0 and 1, which is useful for binary classification problems.

The rectified linear unit (ReLU):

It is a popular choice in neural networks. It is defined as f(x)=max(0,x), where x is the input to the function. For any input x, the output of the ReLU function is x if x is positive and 0 if x is negative. This activation function is simple to implement computationally and faster than other non-linear activation function like tanh or sigmoid.

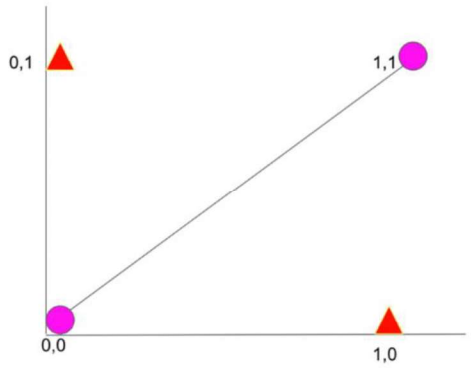


When our data is not linear separable, linear models face problems in approximating whereas it is easy for the neural networks. The hidden layers are used to increase the non-linearity and change the representation of the data for better generalization over the function.

Exercise Questions:

1. Implement two layer Feed Forward Neural Network for XOR Logic Gate with 2-bit Binary Input using Sigmoid activation. Verify the number of learnable parameters in the model.

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0



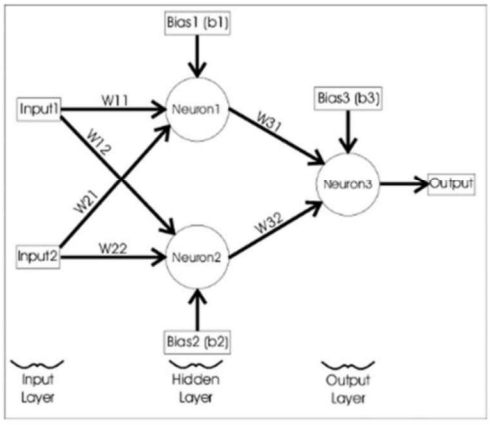
Define the neural network model

Import necessary Libraries
import torch
from matplotlib import pyplot as plt
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import numpy as np

loss_list = []
torch.manual_seed(42)

Step 1: Initialize inputs and expected outputs as per the truth table of XOR
Create the tensors x1,x2 and y.
They are the training examples in the dataset for the XOR
X = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float32)
Y = torch.tensor([0,1,1,0], dtype=torch.float32)

Step 2: Define XORModel class - write constructor and forward function




```
class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        #self.w = torch.nn.Parameter(torch.rand([1]))
        #self.b = torch.nn.Parameter(torch.rand([1]))

        self.linear1 = nn.Linear(2,2,bias=True)
        self.activation1 = nn.Sigmoid()
        self.linear2 = nn.Linear(2,1,bias=True)
        #self.activation2 = nn.ReLU()
    def forward(self, x):
        x = self.linear1(x)
        x = self.activation1(x)
        x = self.linear2(x)
        #x = self.activation2(x)
        return x
```

Step 3: Create DataLoader. Write Dataset class with necessary constructors and methods – len() and getitem()

```
class MyDataset(Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx].to(device), self.Y[idx].to(device)
```

#Create the dataset
full_dataset = MyDataset(X, Y)
batch_size = 1
#Create the dataloaders for reading data - # This provides a way to read the dataset in batches, also shuffle the data
train_data_loader = DataLoader(full_dataset, batch_size=batch_size, shuffle=True)
#Find if CUDA is available to load the model and device on to the available device CPU/GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#Load the model to GPU
model = XORModel().to(device)
print(model)

#Add the criterion which is the MSELoss
loss_fn = torch.nn.MSELoss()
#Optimizers specified in the torch.optim package

optimizer = torch.optim.SGD(model.parameters(), lr=0.03)

```
EPOCHS = 10000
for epoch in range(EPOCHS):
    #print('EPOCH {}:'.format(epoch + 1))

    # Make sure gradient tracking is on, and do a pass over the data
    model.train(True)
    avg_loss = train_one_epoch(epoch)
    loss_list.append(avg_loss)
    #loss_list.append(avg_loss.detach().cpu())

    #print('LOSS train {}'.format(avg_loss))

    if epoch % 1000 == 0:
        print(f'Epoch {epoch}/{EPOCHS}, Loss: {avg_loss}')
```

Training an epoch

```
def train_one_epoch(epoch_index):
    totalloss = 0.
    # use enumerate(training_loader) instead of iter
    for i, data in enumerate(train_data_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn (outputs.flatten(), labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        totalloss += loss.item()

    return totalloss/(len(train_data_loader) * batch_size)
```

Model Inference step

```
for param in model.named_parameters():
    print(param)
Model inference – similar to prediction in ML
input = torch.tensor([0, 1], dtype=torch.float32).to(device)
model.eval()
print("The input is = {}".format(input))
print("Output y predicted ={}".format(model(input)))
#Display the plot
plt.plot(loss_list)
plt.show()
```

Output – Model parameters

```
Model parameters= ('linear1.weight', Parameter containing:
tensor([[ 0.5413,  0.5890],
        [-0.1679,  0.6455]], requires_grad=True))
Model parameters= ('linear1.bias', Parameter containing:
tensor([-0.1505,  0.1357], requires_grad=True))
Model parameters= ('linear2.weight', Parameter containing:
tensor([[-0.3832,  0.3738]], requires_grad=True))
Model parameters= ('linear2.bias', Parameter containing:
tensor([0.5562], requires_grad=True))
```

2. Repeat Qn 1 by modifying the activation function to ReLU.
3. Manually verify the output values by taking system generated values of weights and biases for both Linear1 and Linear2 layers for Qn 1 and apply the transformations to input X and implement the same.
4. Implement Feed Forward Neural Network with two hidden layers for classifying handwritten digits using MNIST dataset. Display the classification accuracy in the form of a Confusion matrix. Verify the number of learnable parameters in the model.

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.