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

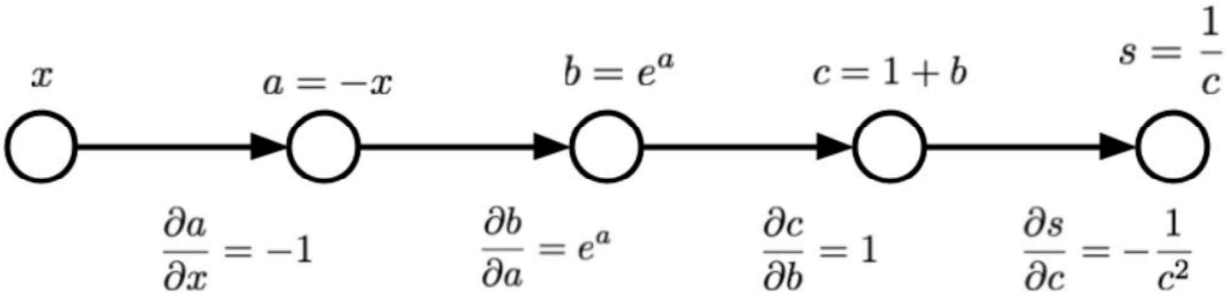
$b(a) = e^a$

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:

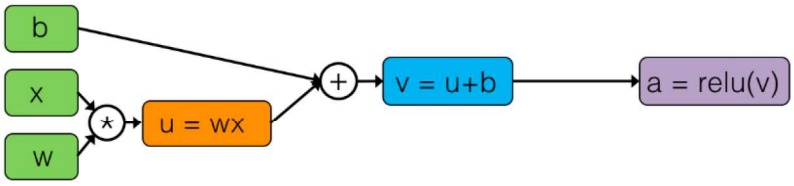
1. Draw Computation Graph and work out the gradient dz/da by following the path back from z to a and compare the result with the analytical gradient.

   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.

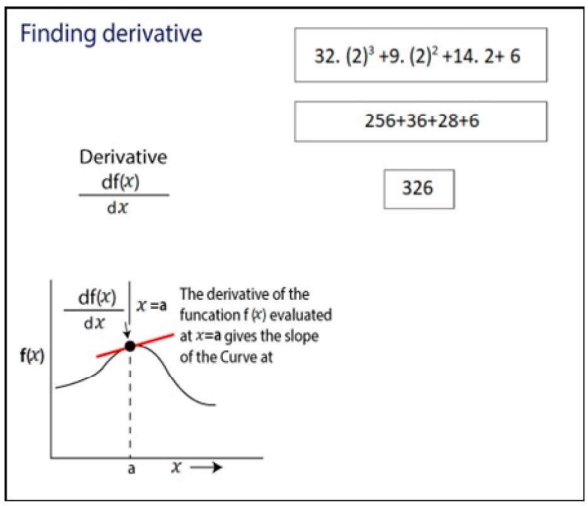
$$8x^4 + 3x^3 + 7x^2 + 6x + 3$$

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

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

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

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

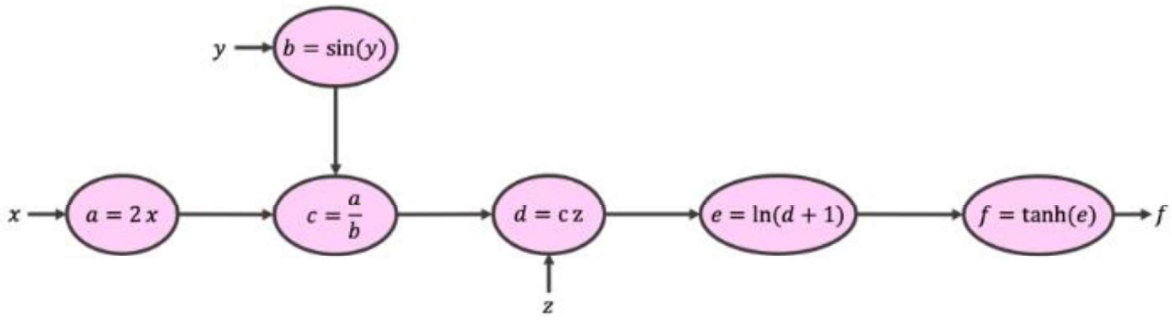



```
import torch
x=torch.tensor(2.0, requires_grad=True)
y=8*x**4+3*x**3+7*x**2+6*x+3
y.backward()
x.grad
```

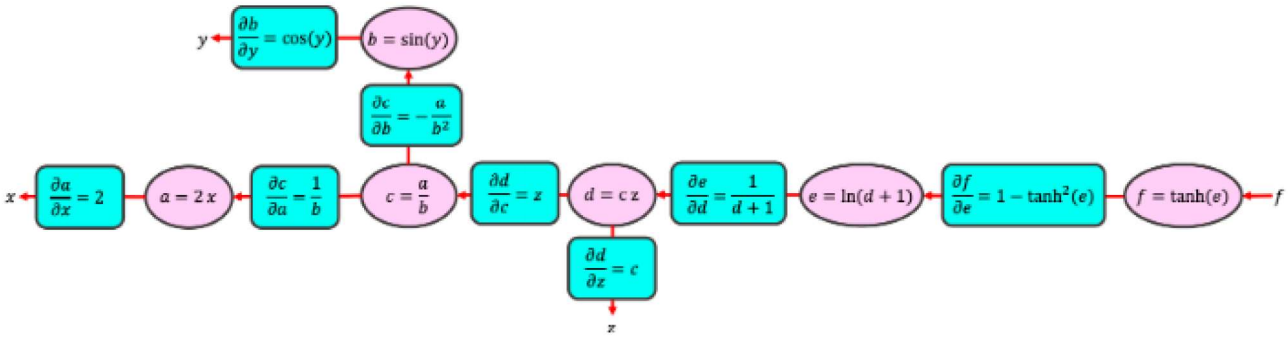
tensor(326.)

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

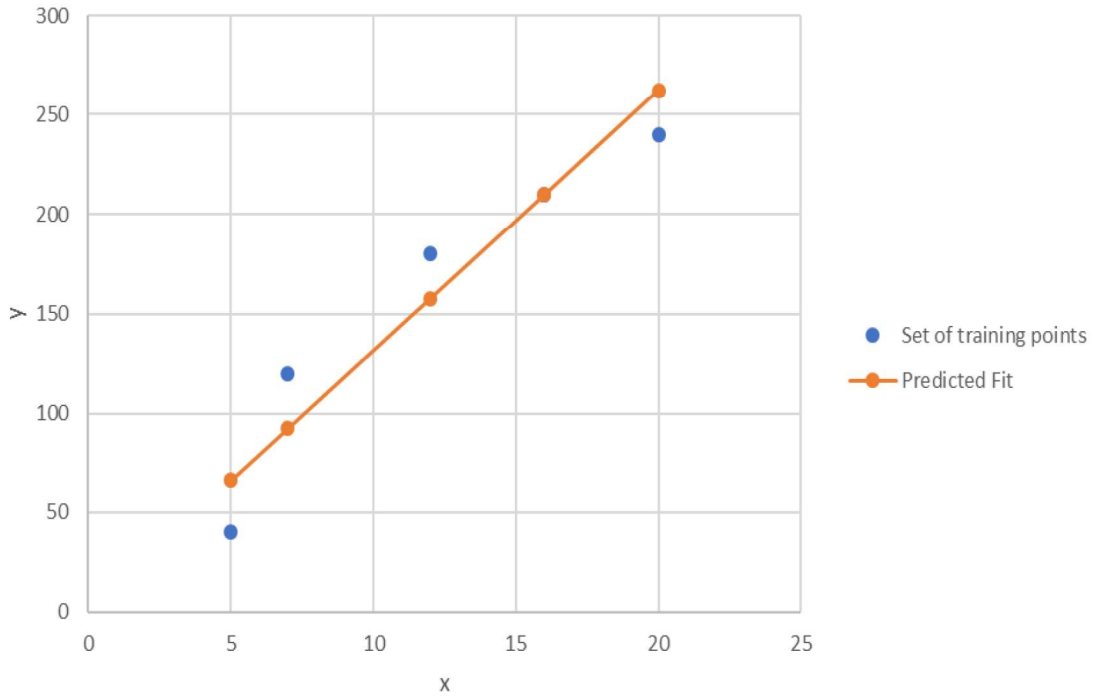
Linear regression is a linear model, a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x).

- 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

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.