22AIE304 Deep Learning Lab Sheet 1

Fifth Semester BTech CSE(AI)

Department of Computer Science and Engineering

Amrita School of Computing

Exercise 1: Refresh NumPy

Name: Girish S

Roll: AM.EN.U4AIE22044

```
import numpy as np
import cv2
import torch as pt
from sklearn.linear_model import LogisticRegression
```

- Shape and type of the array

```
A = np.array([1, 2, 3, 4])
A
```

```
array([1, 2, 3, 4])
```

```
A.shape
```

```
(4,)
```

```
A.dtype
```

```
dtype('int64')
```

- Access specific elements of a 2D NumPy array

```
mat = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(mat)
print("Middle Element: ", mat[1, 1])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Middle Element:  5
```

- Show how to slice a NumPy array to get all rows, but only the first two columns.

```
B = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
B
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
B[:, 0:2]
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
```

- Reshape a 1D array of size 12 into a 2D array with 3 rows and 4 columns?

```
C = np.arange(12)
C
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
D = C.reshape((3, 4))
D
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

- Perform matrix multiplication between two 2D arrays using NumPy.

```
E = B[:, 0:3]
E
```

```
array([[ 1,  2,  3],
       [ 5,  6,  7],
       [ 9, 10, 11],
       [13, 14, 15]])
```

```
F = np.dot(E, D)
F
```

```
array([[ 32,  38,  44,  50],
       [ 80,  98, 116, 134],
       [128, 158, 188, 218],
       [176, 218, 260, 302]])
```

- Compute the mean, median, and standard deviation of a NumPy array

```
np.mean(F), np.median(F), np.std(F)
```

```
(140.0, 131.0, 79.93747556684536)
```

- Perform vertical and horizontal stacking in NumPy

```
G = np.array([5, 6, 7, 8])
G
```

```
array([5, 6, 7, 8])
```

```
np.vstack((A, G))
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
np.hstack((A, G))
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

- Flatten a 3 x 4 NumPy array into a 1D array.

```
D
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
D.reshape(12)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

- Generate a random array of size n with values drawn from a normal distribution

```
n = int(input("Enter the size of the Array: "))
```

```
Enter the size of the Array: 10
```

```
H = np.random.standard_normal(n)
H
```

```
array([ 0.39133488, -0.38747106,  0.51723933,  0.78086399, -0.04963732,
        1.23235815,  0.05614619,  0.07481557, -0.5378615 , -0.97717721])
```

- Perform element-wise addition, subtraction, multiplication, and division

```
B
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

```
B - 5
```

```
array([[-4, -3, -2, -1],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
B+5
```

```
array([[ 6,  7,  8,  9],
       [10, 11, 12, 13],
       [14, 15, 16, 17],
       [18, 19, 20, 21]])
```

```
B*5
```

```
array([[ 5, 10, 15, 20],
       [25, 30, 35, 40],
       [45, 50, 55, 60],
       [65, 70, 75, 80]])
```

```
B/5
```

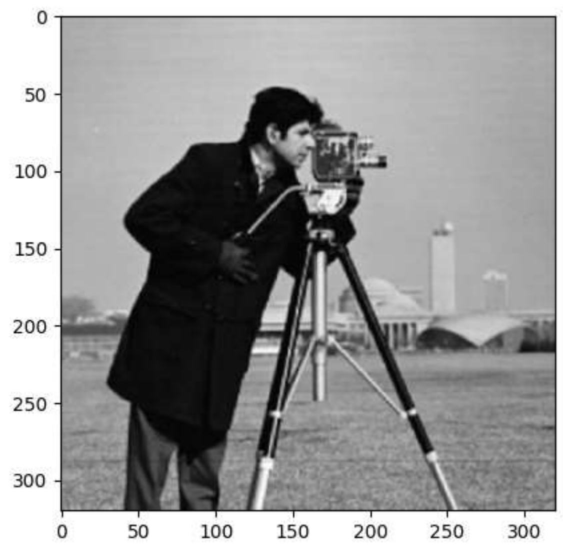
```
array([[0.2, 0.4, 0.6, 0.8],
       [1. , 1.2, 1.4, 1.6],
       [1.8, 2. , 2.2, 2.4],
       [2.6, 2.8, 3. , 3.2]])
```

Exercise 2: Practice Open CV

2.1 Load an image using OpenCV and display it using Matplotlib

```
import cv2
from google.colab.patches import cv2_imshow
from matplotlib import pyplot as plt


img = cv2.imread('/content/cameraman.jpg', -1)
plt.imshow(img)
```

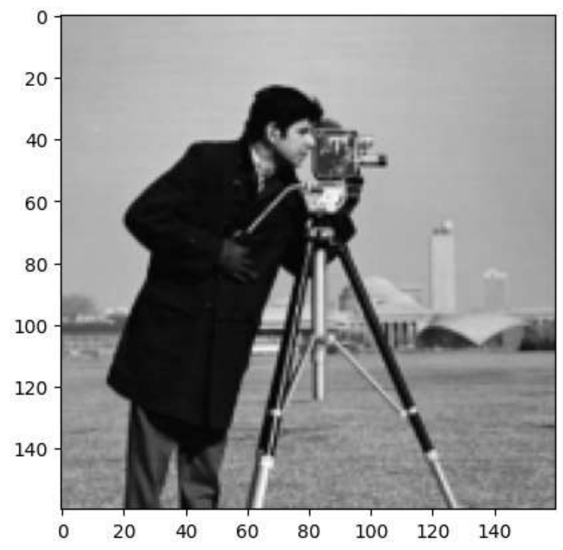
```
<matplotlib.image.AxesImage at 0x795e981b3130>
```



2.2 Resize the image to half of its original size using OpenCV

```
height, width = img.shape[:2]

resized_img = cv2.resize(img, (width//2, height//2))

plt.imshow(resized_img)
```

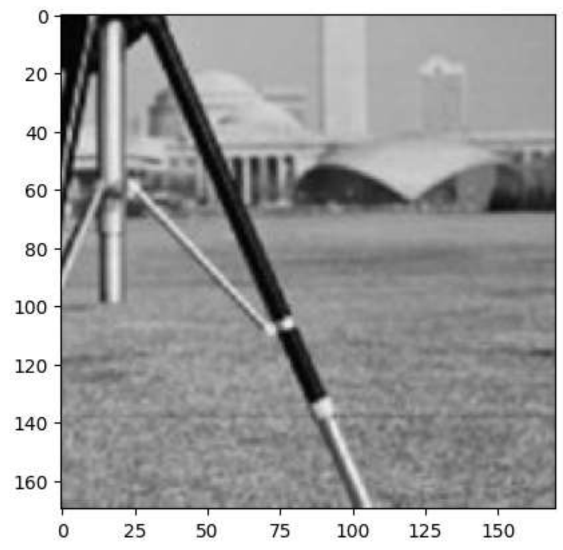
```
<matplotlib.image.AxesImage at 0x795e7dd90880>
```



2.3 Crop a specific region (e.g., the top-left quarter) of an image using NumPy slicing in OpenCV.

```
cropped = img[150:, 150:] # bottom right 150*150 pixels
plt.imshow(cropped)
```

```
<matplotlib.image.AxesImage at 0x795e7dd96b30>
```

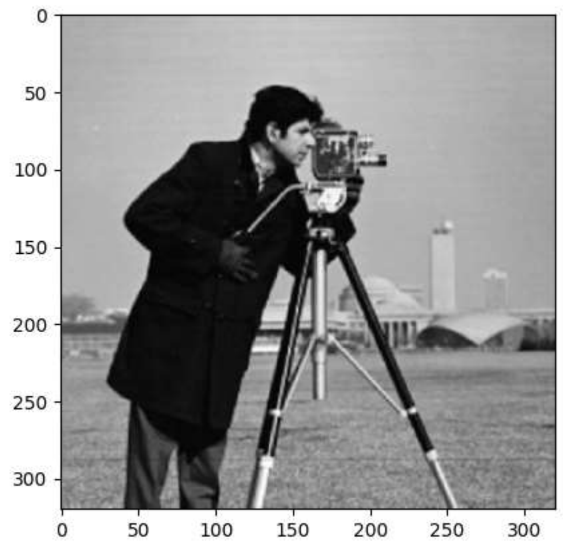


2.4 Convert an image from BGR (OpenCV default) to grayscale.

```
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # it is already gray scale thus no change
```

```
plt.imshow(img2, cmap='gray')
```

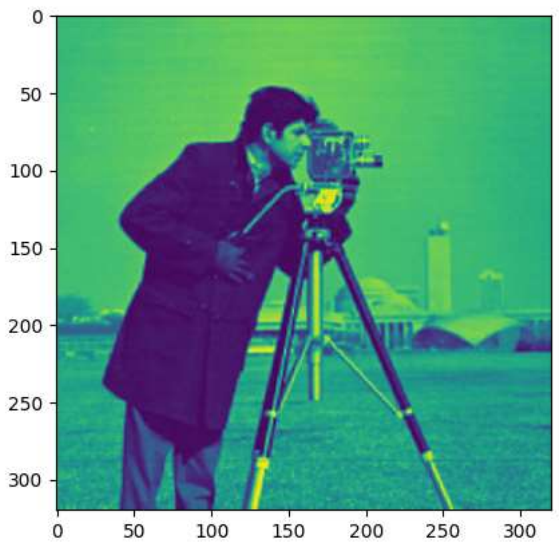
```
<matplotlib.image.AxesImage at 0x795e7de2fac0>
```



2.5 Display the image using Matplotlib instead of OpenCV

```
plt.imshow(img2) # this exercise is done in google colab,
# thus i cant display using cv. all other snippets are displayed using matplotlib which works with colab
```

```
<matplotlib.image.AxesImage at 0x795e7da33b80>
```



2.6 Find the dimensions (width, height, and channels) of an image using OpenCV

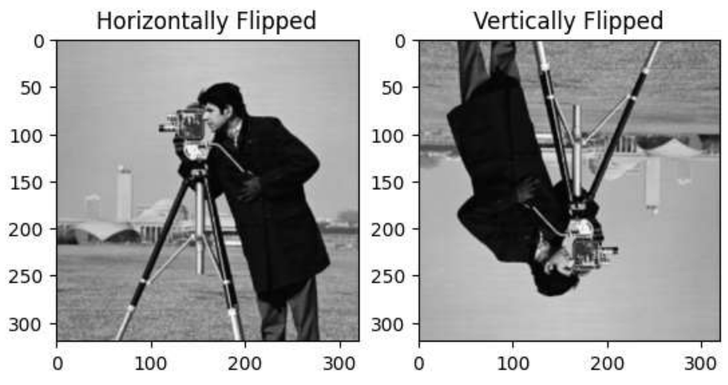
```
print("(Height, width, channels) :", img.shape)
```

```
(Height, width, channels) : (320, 320, 3)
```

2.7 Flip the image horizontally and vertically using OpenCV.

```
horizontal = cv2.flip(img, 1)
vertical = cv2.flip(img, 0)

plt.subplot(1, 2, 1)
plt.imshow(horizontal)
plt.title("Horizontally Flipped")

plt.subplot(1, 2, 2)
plt.imshow(vertical)
plt.title("Vertically Flipped")
```

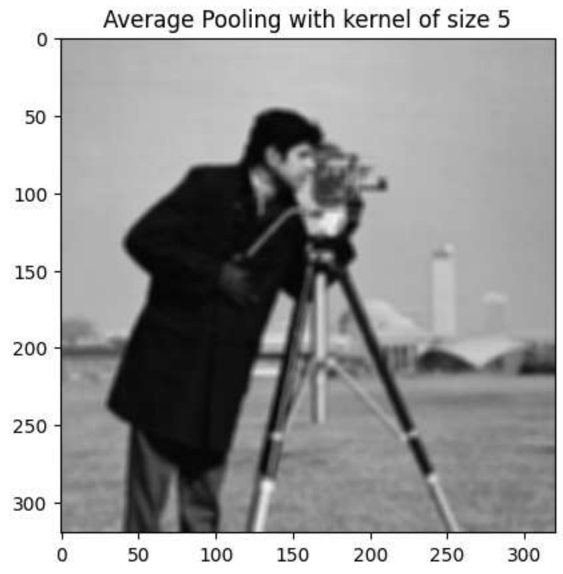
```
Text(0.5, 1.0, 'Vertically Flipped')
```



2.8 Apply a 5x5 averaging (box) filter to smoothen the image.

```
blur = cv2.blur(img, (5, 5))

plt.imshow(blur)
plt.title("Average Pooling with kernel of size 5")
```

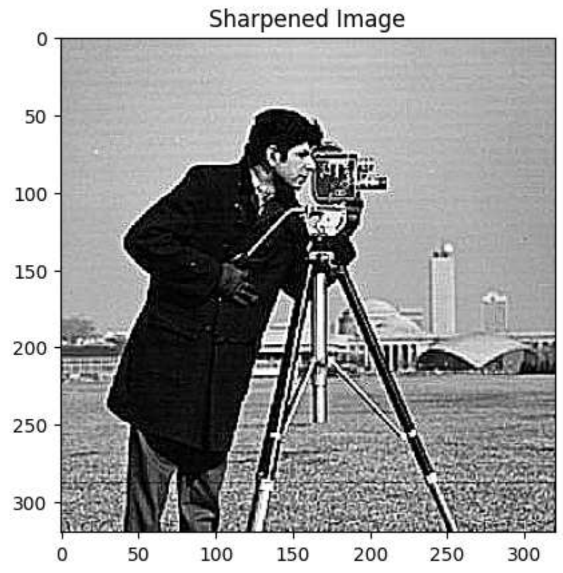
```
Text(0.5, 1.0, 'Average Pooling with kernel of size 5')
```



2.9 Apply a sharpening kernel to enhance the edges and details in the image.

```
import numpy as np


kernel = np.array([[-1, -1, -1],
                   [-1, 9, -1],
                   [-1, -1, -1]])

sharpened = cv2.filter2D(img, -1, kernel)

plt.imshow(sharpened)
plt.title("Sharpened Image")
```

```
Text(0.5, 1.0, 'Sharpened Image')
```

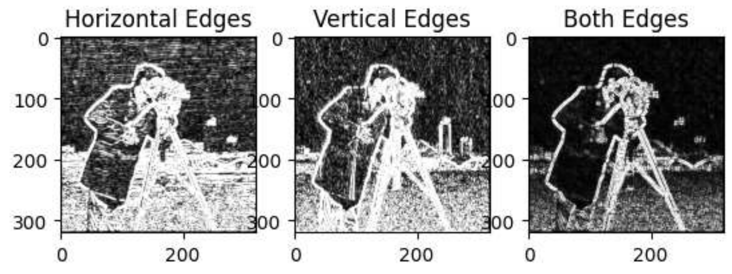


2.10 Apply the Sobel operator to detect edges in both the horizontal and vertical directions.

```
vertical = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5) # 5x5 kernel with sobel on x axis
horizontal= cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5) # 5x5 kernel with sobel on Y axis
combined = cv2.Sobel(img, cv2.CV_64F, 1, 1, ksize=5) # 5x5 kernel with both horizontal and vertical edge detection

vertical = cv2.convertScaleAbs(vertical)
```

```
horizontal = cv2.convertScaleAbs(horizontal)
combined = cv2.convertScaleAbs(combined)

plt.subplot(1, 3, 1)
plt.imshow(horizontal)
plt.title("Horizontal Edges")

plt.subplot(1, 3, 2)
plt.imshow(vertical)
plt.title("Vertical Edges")

plt.subplot(1, 3, 3)
plt.imshow(combined)
plt.title("Both Edges")
```

```
Text(0.5, 1.0, 'Both Edges')
```



Exercise 3: Practice PyTorch

3.1 Create two random 3x3 tensors and perform matrix multiplication. Compute the matrix product and use Py Torch's autograd to calculate the gradient of the result with respect to one of the input tensors.

```
import torch

A = torch.rand((3, 3), requires_grad=True)
B = torch.rand((3, 3))

print("A:", A, "\n\n\nB:", B)

C = torch.matmul(A, B)

print("C:", )

C.sum().backward()

print("Gradient of C with respect to A:", A.grad)
```

```
A: tensor([[0.1033, 0.5410, 0.8568],
        [0.0327, 0.3485, 0.5990],
        [0.9345, 0.2973, 0.0303]], requires_grad=True)


B: tensor([[0.2206, 0.3604, 0.9638],
        [0.8284, 0.5824, 0.4280],
        [0.2826, 0.6663, 0.2858]])
C:
Gradient of C with respect to A: tensor([[1.5448, 1.8388, 1.2348],
        [1.5448, 1.8388, 1.2348],
        [1.5448, 1.8388, 1.2348]])
```

3.2 Perform element-wise operations on tensors with broadcasting. Create a 3x1 tensor and a 1x3 tensor. Use broadcasting to add them and multiply the result by another tensor of shape 3x3.Explore how broadcasting works in PyTorch and understand how it simplifies tensor operations.

```
A = torch.tensor([[1], [2], [3]])
B = torch.tensor([[1, 2, 3]])
```

```
summer = A + B
C = torch.rand((3, 3))


res = summer * C

print("Broadcasted Result:\n", summer)
print("Final Result (after multiplication):\n", res)
```

```
Broadcasted Result:
 tensor([[2, 3, 4],
        [3, 4, 5],
        [4, 5, 6]])
Final Result (after multiplication):
 tensor([[6.8771e-01, 9.4050e-02, 1.6746e+00],
        [1.3490e-03, 2.1248e-01, 2.7833e+00],
        [2.4648e+00, 3.5443e-02, 1.9430e+00]])
```

3.3 Create a 2D tensor of shape (6, 4) and reshape it into a tensor of shape (3, 8). Extract specific slices from the reshaped tensor (e.g., select all rows but only the first two columns).

```
A = torch.rand((6, 4))
print(A)


B = A.reshape((3, 8))
print("\n\nB:\n", B)

print("\n\nFirst 2 Columns:\n", B[:, 0:2])
```

```
tensor([[0.9717, 0.1016, 0.4277, 0.1567],
        [0.5278, 0.8656, 0.3436, 0.4371],
        [0.9502, 0.0273, 0.9214, 0.2056],
        [0.2599, 0.9796, 0.3809, 0.9109],
        [0.4093, 0.8402, 0.0587, 0.9325],
        [0.2341, 0.7641, 0.4358, 0.1623]])


B:
 tensor([[0.9717, 0.1016, 0.4277, 0.1567, 0.5278, 0.8656, 0.3436, 0.4371],
        [0.9502, 0.0273, 0.9214, 0.2056, 0.2599, 0.9796, 0.3809, 0.9109],
        [0.4093, 0.8402, 0.0587, 0.9325, 0.2341, 0.7641, 0.4358, 0.1623]])


First 2 Columns:
 tensor([[0.9717, 0.1016],
        [0.9502, 0.0273],
        [0.4093, 0.8402]])
```

3.4 Create a NumPy array, convert it into a PyTorch tensor, perform some operations (e.g., multiplication by a scalar), and convert the result back to a NumPy array.

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Numpy Array:\n", arr)

tensor = torch.from_numpy(arr)
print("\n\nTensor:\n", tensor)

tensor = tensor*2
print("\n\n\nTensor after Multiplication:\n", tensor)

array = tensor.numpy()
print("\n\n\nNumpy Array Again:\n", array)
```

```
Numpy Array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]


Tensor:
 tensor([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])



    Tensor after Multiplication:
     tensor([[ 2,  4,  6],
            [ 8, 10, 12],
            [14, 16, 18]])



    Numpy Array Again:
     [[ 2  4  6]
     [ 8 10 12]
     [14 16 18]]
```

3.5 Initialize a 5x5 tensor with random values sampled from a uniform distribution between 0 and 1. Initialize another 5x5 tensor with random values sampled from a normal distribution with a mean of 0 and a standard deviation of 1. Multiply the two tensors elementwise. Compute the mean and standard deviation of the resulting tensor. Reshape the result into a 1D tensor of size 25. Compute the sum of all elements in the reshaped tensor.

```
tensor1 = torch.rand((5, 5))
print("Tensor 5x5 Uniform Dist:\n", tensor1)

tensor2 = torch.randn((5, 5))
print("\n\n\nTensor 5x5 Normal Dist:\n", tensor2)

tensorprod = tensor1 * tensor2
print("\n\n\nTensor Product:\n", tensorprod)

print("\n\n\nTensor Mean:\n", tensorprod.mean())
print("\n\n\nTensor Standard Deviation:\n", tensorprod.std())

reshaped = tensorprod.view(25)
print("\n\n\nTensor Reshaped to 1x25:\n", reshaped)

print("\n\n\nSum of Tensor:\n", reshaped.sum())
```

```
    Tensor 5x5 Uniform Dist:
     tensor([[0.2001, 0.6836, 0.8601, 0.2343, 0.1390],
            [0.1780, 0.0196, 0.3832, 0.3378, 0.0779],
            [0.7345, 0.5200, 0.6766, 0.3017, 0.1582],
            [0.3196, 0.3590, 0.7084, 0.9595, 0.4879],
            [0.6255, 0.2763, 0.9708, 0.2844, 0.3087]])



    Tensor 5x5 Normal Dist:
     tensor([[-2.0025,  0.6068, -0.0582,  0.5855,  0.1064],
            [-1.7789,  0.6003, -0.9258,  0.0923, -0.8397],
            [ 0.0166, -0.1784, -0.8743, -0.9429, -0.3562],
            [ 0.8893,  2.5445,  0.9925, -2.0126, -0.4720],
            [-0.1345,  0.3718,  0.3967,  0.6509, -0.6923]])



    Tensor Product:
     tensor([[-0.4008,  0.4148, -0.0500,  0.1372,  0.0148],
            [-0.3167,  0.0118, -0.3548,  0.0312, -0.0654],
            [ 0.0122, -0.0928, -0.5916, -0.2844, -0.0563],
            [ 0.2842,  0.9134,  0.7030, -1.9311, -0.2303],
            [-0.0841,  0.1027,  0.3851,  0.1851, -0.2137]])



    Tensor Mean:
     tensor(-0.0591)



    Tensor Standard Deviation:
     tensor(0.5155)
```

```
Tensor Reshaped to 1x25:
 tensor([-0.4008,  0.4148, -0.0500,  0.1372,  0.0148, -0.3167,  0.0118, -0.3548,
         0.0312, -0.0654,  0.0122, -0.0928, -0.5916, -0.2844, -0.0563,  0.2842,
         0.9134,  0.7030, -1.9311, -0.2303, -0.0841,  0.1027,  0.3851,  0.1851,
        -0.2137])



Sum of Tensor:
 tensor(-1.4764)
```

Exercise 4:

4.1 Build a function that returns the sigmoid of a real number x. Use math.exp(x) for the exponential function.

> **Note:** sigmoid(x)=1/(1+e^-x) is sometimes also known as the logistic function. It is a non-linear function used in Machine Learning (Logistic Regression) and Deep Learning.
>
> use **np.exp()** to Implement the sigmoid function using NumPy. see why np.exp() is preferable to math.exp().

```
def sigmoid(x):
  return 1/(1+np.exp(-x))

print("Sigmoid of Array[1, 2, 3]: ", sigmoid(np.array([1, 2, 3])))
# np.exp works with arrays to making it suitable for vectorized inputs while math.exp can only handle scalar inputs
```

```
Sigmoid of Array[1, 2, 3]:  [0.73105858 0.88079708 0.95257413]
```

4.2 Implement the function sigmoid_grad() to compute the gradient of the sigmoid function with respect to its input x. The formula is: sigmoid_derivative(x)=σ'(x)=σ(x)(1−σ(x))

```
def sigmoid_grad(x):
  sig = sigmoid(x)
  return sig*(1-sig)

print("Gradient of Sigmoid of Array[1, 2, 3]: ", sigmoid_grad(np.array([1, 2, 3])))
```

```
Gradient of Sigmoid of Array[1, 2, 3]:  [0.19661193 0.10499359 0.04517666]
```

4.3 Implement image2vector() that takes an input of shape (length, height, 3) and returns a vector of shape (length*height*3, 1).

```
def image2vector(image):
  return image.reshape(-1, 1)

image = np.random.rand(3, 3, 3)
vector = image2vector(image)
print(f"Vector shape: {vector.shape}")
print(f"Vector: \n{vector}")
```

```
Vector shape: (27, 1)
Vector:
[[0.347454  ]
 [0.16950734]
 [0.22481364]
 [0.52178675]
 [0.24321983]
 [0.70051819]
 [0.42896275]
 [0.93891608]
 [0.55539417]
 [0.93642204]
```

```
    [0.60557478]
    [0.9799819 ]
    [0.0463629 ]
    [0.33752564]
    [0.46445997]
    [0.31746285]
    [0.76271906]
    [0.47763752]
    [0.56832232]
    [0.43535842]
    [0.40764585]
    [0.50111977]
    [0.17650402]
    [0.68513553]
    [0.2898987 ]
    [0.2628609 ]
    [0.84933524]]
```

4.4 Implement normalizeRows() to normalize the rows of a matrix. After applying this function to an input matrix x, each row of x should be a vector of unit length (meaning length 1).

```
def normalizeRows(x):
    norm = np.linalg.norm(x, axis=1, keepdims=True)
    return x / norm

x = np.array([[1, 2, 3], [4, 5, 6]])
normalized_x = normalizeRows(x)
print(f"Original matrix:\n{x}")
print(f"Normalized matrix:\n{normalized_x}")
```

```
Original matrix:
[[1 2 3]
 [4 5 6]]
Normalized matrix:
[[0.26726124 0.53452248 0.80178373]
 [0.45584231 0.56980288 0.68376346]]
```

4.5 Implement the L1 and L2 loss functions:

L1 loss is defined as:

$$L_1(\hat{y}, y) = \sum_{i=0}^{m-1} |y^{(i)} - \hat{y}^{(i)}|$$

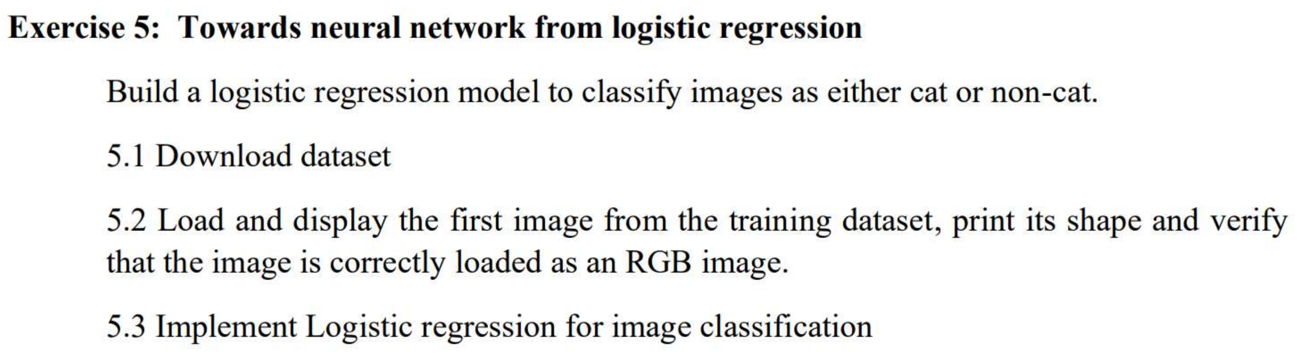
L2 loss is defined as:

$$L_2(\hat{y}, y) = \sum_{i=0}^{m-1} (y^{(i)} - \hat{y}^{(i)})^2$$

```
def l1_loss(y_pred, y_true):
    return torch.mean(torch.abs(y_pred - y_true))

def l2_loss(y_pred, y_true):
    return torch.mean((y_pred - y_true) ** 2)

# Example usage
y_pred = torch.tensor([2.0, 3.0, 4.0])
y_true = torch.tensor([3.0, 3.0, 3.0])

l1 = l1_loss(y_pred, y_true)
l2 = l2_loss(y_pred, y_true)
```

```
print("L1 Loss:", l1.item())
print("L2 Loss:", l2.item())
```

```
L1 Loss: 0.6666666865348816
L2 Loss: 0.6666666865348816
```



Exercise 5: Towards neural network from logistic regression

Build a logistic regression model to classify images as either cat or non-cat.

5.1 Download dataset

5.2 Load and display the first image from the training dataset, print its shape and verify that the image is correctly loaded as an RGB image.

5.3 Implement Logistic regression for image classification

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from tensorflow.keras.datasets import cifar10


(X_train, y_train), (X_test, y_test) = cifar10.load_data()


cat_label = 3
y_train_cat = (y_train == cat_label).astype(int).flatten()  # 1 for cat, 0 for non-cat
y_test_cat = (y_test == cat_label).astype(int).flatten()


X_train_flat = X_train.reshape(X_train.shape[0], -1) / 255.0
X_test_flat = X_test.reshape(X_test.shape[0], -1) / 255.0


X_train_tensor = torch.tensor(X_train_flat, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_cat, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_flat, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_cat, dtype=torch.float32)


class LogisticRegressionModel(nn.Module):
    def __init__(self, input_size):
        super(LogisticRegressionModel, self).__init__()
        self.linear = nn.Linear(input_size, 1)

    def forward(self, x):
        return torch.sigmoid(self.linear(x))

input_size = X_train_tensor.shape[1]
model = LogisticRegressionModel(input_size)
criterion = nn.BCELoss()  # Binary Cross-Entropy Loss
optimizer = optim.SGD(model.parameters(), lr=0.01)


num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    outputs = model(X_train_tensor).squeeze()
    loss = criterion(outputs, y_train_tensor)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()
```

```
        if (epoch+1) % 1 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')


    model.eval()
    with torch.no_grad():
        y_pred_prob = model(X_test_tensor).squeeze()
        y_pred = (y_pred_prob >= 0.5).float()  # Convert probabilities to binary output

    # Calculate accuracy
    accuracy = (y_pred == y_test_tensor).float().mean().item()
    print("Accuracy:", accuracy)

    # Display the first image from the test dataset
    plt.imshow(X_test[0])
    plt.title(f"True label: {y_test[0][0]}, Predicted label: {y_pred[0].item()}")
    plt.axis('off')
    plt.show()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
Epoch [1/10], Loss: 0.7460
Epoch [2/10], Loss: 0.3443
Epoch [3/10], Loss: 0.3342
Epoch [4/10], Loss: 0.3302
Epoch [5/10], Loss: 0.3288
Epoch [6/10], Loss: 0.3283
Epoch [7/10], Loss: 0.3281
Epoch [8/10], Loss: 0.3279
Epoch [9/10], Loss: 0.3278
Epoch [10/10], Loss: 0.3276
Accuracy: 0.8999999761581421
```