HW1 - 15 Points

- **You have to submit two files for this part of the HW**

> (1) ipynb (colab notebook) and
> (2) pdf file (pdf version of the colab file).**

- **Files should be named as follows**:

> FirstName_LastName_HW_1**

```
!nvidia-smi
```

```
Thu Jan 25 20:16:12 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-SXM4-40GB          Off | 00000000:00:04.0 Off |                    0 |
| N/A   31C    P0              47W / 400W |      2MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

```
import torch
import time
```

Q1 : Create Tensor (1/2 Point)

Create a torch Tensor of shape (5, 3) which is filled with zeros. Modify the tensor to set element (0, 2) to 10 and element (2, 0) to 100.

```
my_tensor = torch.zeros(5,3)
```

```
my_tensor.shape
```

```
torch.Size([5, 3])
```

```
my_tensor
```

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

```
# Manually set the value at the first row and third column to 10,
# and the value at the third row and first column to 100 in the tensor named "my_tensor".

my_tensor[(0,2),(2,0)] = torch.tensor([10.,100.])
```

```
my_tensor
```

```
tensor([[  0.,   0.,  10.],
        [  0.,   0.,   0.],
        [100.,   0.,   0.],
        [  0.,   0.,   0.],
        [  0.,   0.,   0.]])
```

Q2: Reshape tensor (1/2 Point)

You have following tensor as input:

`x=torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])`

Using only reshaping functions (like view, reshape, transpose, permute), you need to get at the following tensor as output:

```
tensor([[ 0,  4,  8, 12, 16, 20],
        [ 1,  5,  9, 13, 17, 21],
        [ 2,  6, 10, 14, 18, 22],
        [ 3,  7, 11, 15, 19, 23]])
```

```
x=torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])
```

```
x = x.view(4,6).reshape(6,4).T
x
```

```
tensor([[ 0,  4,  8, 12, 16, 20],
        [ 1,  5,  9, 13, 17, 21],
        [ 2,  6, 10, 14, 18, 22],
        [ 3,  7, 11, 15, 19, 23]])
```

Q3: Slice tensor (1Point)

- Slice the tensor x to get the following

> - last row of x
> - fourth column of x
> - first three rows and first two columns - the shape of subtensor should be (3,2)
> - odd valued rows and columns

```
x = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 8, 10], [11, 12, 13, 14, 15]])
x
```

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

```
x.shape
```

```
torch.Size([3, 5])
```

```
# Student Task: Retrieve the last row of the tensor 'x'
# Hint: Negative indexing can help you select rows or columns counting from the end of the tensor.
# Think about how you can select all columns for the desired row.
last_row = x[2,:]
last_row
```

```
tensor([11, 12, 13, 14, 15])
```

```
# Student Task: Retrieve the fourth column of the tensor 'x'
# Hint: Pay attention to the indexing for both rows and columns.
# Remember that indexing in Python starts from zero.
fourth_column = x[:,3]
fourth_column
```

```
tensor([ 4,  8, 14])
```

```
# Student Task: Retrieve the first 3 rows and first 2 columns from the tensor 'x'.
# Hint: Use slicing to extract the required subset of rows and columns.
first_3_rows_2_columns = x[:3,:2]
first_3_rows_2_columns
```

```
tensor([[ 1,  2],
        [ 6,  7],
```

```
        [11, 12]])
```

```
# Student Task: Retrieve the rows and columns with odd-indexed positions from the tensor 'x'.
# Hint: Use stride slicing to extract the required subset of rows and columns with odd indices.
odd_valued_rows_columns =  x[::2,::2]
odd_valued_rows_columns
```

```
tensor([[ 1,  3,  5],
        [11, 13, 15]])
```

Q4 -Normalize Function (1/2 Points)

Write the function that normalizes the columns of a matrix. You have to compute the mean and standard deviation of each column. Then for each element of the column, you subtract the mean and divide by the standard deviation.

```
# Given Data
x = [[ 3,  60,  100, -100],
     [ 2,  20,  600, -600],
     [-5,  50,  900, -900]]
```

```
# Convert to PyTorch Tensor and set to float
X = torch.tensor(x)
X= X.float()
```

```
# Print shape and data type for verification
print(X.shape)
print(X.dtype)
```

```
torch.Size([3, 4])
torch.float32
```

```
# Compute and display the mean and standard deviation of each column for reference
X.mean(axis = 0)
X.std(axis = 0)
```

```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

```
a=X-X.mean(axis=0)
a
```

```
tensor([[   3.0000,   16.6667, -433.3333,  433.3333],
        [   2.0000,  -23.3333,   66.6667,  -66.6667],
        [  -5.0000,    6.6667,  366.6667, -366.6667]])
```

```
X.std(axis = 0)
```

```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

```
X.mean(axis=1)
```

```
tensor([15.7500,  5.5000, 11.2500])
```

```
X.mean(axis=0)
```

```
tensor([   0.0000,   43.3333,  533.3333, -533.3333])
```

- Your task starts here
- Your normalize_matrix function should take a PyTorch tensor x as input.
- It should return a tensor where the columns are normalized.
- After implementing your function, use the code provided to verify if the mean for each column in Z is close to zero and the standard deviation is 1.

```
def normalize_matrix(x):
  # Calculate the mean along each column (think carefully , you will take mean along axis = 0 or 1)
  mean = x.mean(axis=0)

  # Calculate the standard deviation along each column
  std = x.std(axis=0)

  # Normalize each element in the columns by subtracting the mean and dividing by the standard deviation
  y = (x-mean)/std

  return y  # Return the normalized matrix
```

```
Z = normalize_matrix(X)
Z
```

```
tensor([[ 0.6882,  0.8006, -1.0722,  1.0722],
        [ 0.4588, -1.1209,  0.1650, -0.1650],
        [-1.1471,  0.3203,  0.9073, -0.9073]])
```

```
Z.mean(axis = 0)
```

```
tensor([ 0.0000e+00,  4.9671e-08,  3.9736e-08, -3.9736e-08])
```

Q5: In-place vs. Out-of-place Operations (1 Point)

1. Create a tensor `A` with values `[1, 2, 3]`.
2. Perform an in-place addition (use `add_` method) of `5` to tensor `A`.
3. Then, create another tensor `B` with values `[4, 5, 6]` and perform an out-of-place addition of `5`.

Print the memory addresses of `A` and `B` before and after the operations to demonstrate the difference in memory usage. Provide explanation

```
A = torch.tensor([1, 2, 3])
print('Original memory address of A:', id(A))
A.add_(5)
print('Memory address of A after in-place addition:', id(A))
print('A after in-place addition:', A)

B = torch.tensor([4, 5, 6])
print('Original memory address of B:', id(B))
B = B + 5
print('Memory address of B after out-of-place addition:', id(B))
print('B after out-of-place addition:', B)
```

```
Original memory address of A: 132148288660368
Memory address of A after in-place addition: 132148288660368
A after in-place addition: tensor([6, 7, 8])
Original memory address of B: 132151546133088
Memory address of B after out-of-place addition: 132148272363168
B after out-of-place addition: tensor([ 9, 10, 11])
```

Provide Explanation for above question here :

- We Created a Tensor A which is stored in CPU memory place and can be found using id function
- As noticed, we used in place addition to add 5 to the tensor. As it is inplace operation, The A will be written over, so memory will be in the same place
- Where as in B+5 will be stored in new memory and then B is assigned to that Tenor, Hence the new place in memory

Q6: Tensor Broadcasting (1 Point)

1. Create two tensors `X` with shape `(3, 1)` and `Y` with shape `(1, 3)`. Perform an addition operation on `X` and `Y`.
2. Explain how broadcasting is applied in this operation by describing the shape changes that occur internally.

```
X = torch.randn(3,1)
Y = torch.randn(1,3)
print(f"{X},{Y}")
print('Original shapes:', X.shape, Y.shape)
result = X+Y
print('Result:', result)
print('Result shape:', result.shape)
```

```
tensor([[ 0.4998],
        [-0.2448],
        [-0.0642]]),tensor([[-1.2348, -1.3550,  0.0977]])
Original shapes: torch.Size([3, 1]) torch.Size([1, 3])
Result: tensor([[-0.7350, -0.8552,  0.5975],
        [-1.4796, -1.5999, -0.1471],
        [-1.2989, -1.4192,  0.0336]])
Result shape: torch.Size([3, 3])
```

Provide Explanation for above question here :

- Creating random tensors of 1 dimesnsion with shape as (3,1) and (1,3)
- Adding operation on X and Y which brodcasts the new matrix (3,3)
- In here , The only column in X is added with first element in only row Y matrix. This gives 3 elements
- The same with next, The only column in X is added with Second element in only row Y matrix. and then with Third element in Y matrix. Giving us 3 and 3 elements respectively.
- All the 3,3,3 elements formed will be a new resulting matrix of (3,3)

Q7: Linear Algebra Operations (1 Point)

1. Create two matrices `M1` and `M2` of compatible shapes for matrix multiplication. Perform the multiplication and print the result.
2. Then, create two vectors `V1` and `V2` and compute their dot product.

```
M1 = torch.randn(5,1)
M2 = torch.randn(1,5)
mat_multiplication =  torch.mm(M1,M2)
print('Matrix multiplication result:', mat_multiplication)

V1 = torch.randn(6,)
V2 =  torch.randn(6,)
dot_product =  torch.dot(V1,V2)
print('Dot product:', dot_product)
```

```
Matrix multiplication result: tensor([[-0.1088, -0.9379,  0.2487, -0.1640,  0.0985],
        [ 0.2762,  2.3807, -0.6312,  0.4162, -0.2500],
        [ 0.0570,  0.4910, -0.1302,  0.0858, -0.0516],
        [-0.1074, -0.9257,  0.2454, -0.1618,  0.0972],
        [-0.0814, -0.7018,  0.1861, -0.1227,  0.0737]])
Dot product: tensor(-4.1599)
```

Q8: Manipulating Tensor Shapes (1 Point)

Given a tensor `T` with shape `(2, 3, 4)`, demonstrate how to

1. reshape it to `(3, 8)` using view,
2. reshape it to `(4, 2, 3` using reshape,
3. transpose the first and last dimensions using permute.
4. explain what is the difference between reshape and view

```
T = torch.rand(2, 3, 4)
print(T)
T_view = T.view(3,8)
print('T_view shape:', T_view.shape)
print(T_view)

T_reshape =  T.reshape(4,2,3)
print(T_reshape)
print('T_reshape shape:', T_reshape.shape)

T_permute = T.permute(2,1,0)
print(T_permute)
print('T_permute shape:', T_permute.shape)
```

```
tensor([[[0.3454, 0.1919, 0.3620, 0.7798],
         [0.3681, 0.0226, 0.0707, 0.7579],
         [0.1362, 0.0631, 0.6055, 0.3075]],

        [[0.2639, 0.4331, 0.8428, 0.5188],
         [0.3843, 0.1074, 0.8588, 0.3858],
         [0.6176, 0.9282, 0.4978, 0.8195]]])
T_view shape: torch.Size([3, 8])
tensor([[0.3454, 0.1919, 0.3620, 0.7798, 0.3681, 0.0226, 0.0707, 0.7579],
        [0.1362, 0.0631, 0.6055, 0.3075, 0.2639, 0.4331, 0.8428, 0.5188],
        [0.3843, 0.1074, 0.8588, 0.3858, 0.6176, 0.9282, 0.4978, 0.8195]])
tensor([[[0.3454, 0.1919, 0.3620],
         [0.7798, 0.3681, 0.0226]],

        [[0.0707, 0.7579, 0.1362],
         [0.0631, 0.6055, 0.3075]],

        [[0.2639, 0.4331, 0.8428],
         [0.5188, 0.3843, 0.1074]],

        [[0.8588, 0.3858, 0.6176],
         [0.9282, 0.4978, 0.8195]]])
T_reshape shape: torch.Size([4, 2, 3])
tensor([[[0.3454, 0.2639],
         [0.3681, 0.3843],
         [0.1362, 0.6176]],

        [[0.1919, 0.4331],
         [0.0226, 0.1074],
         [0.0631, 0.9282]],

        [[0.3620, 0.8428],
         [0.0707, 0.8588],
         [0.6055, 0.4978]],

        [[0.7798, 0.5188],
         [0.7579, 0.3858],
         [0.3075, 0.8195]]])
T_permute shape: torch.Size([4, 3, 2])
```

Provide Explanation for above question here :

- Creating a tensor with shape (2,3,4) means 2 dimensions and each dimension with 3 rows and 4 columns
- By view function we rearrange the tensor with 1 dimension of 3 rows and 8 columns
- By reshape function we are changing the shape with 4 dimensions with matrices of 2 rows and 3 columns
- By permute function we can re arrange the elements in the tensor from have (2,3,4) to (4,3,2) resultiung tensor being 4 dimensions of 3x2 matrices

Q9: Tensor Concatenation and Stacking (1 Point)

Create tensors `C1` and `C2` both with shape (2, 3).

1. Concatenate them along dimension 0 and then along dimension 1. Print the shape of the resulting tensor.
2. Afterwards, stack the same tensors alomng dimension 0 and print the shape of the resulting tensor.
3. What is the difference between stacking and concatinating.

```
C1 = torch.rand(2, 3)
C2 = torch.rand(2, 3)
concatenated_dim0 =  torch.cat((C1,C2),dim=0)
print('Concatenated along dimension 0:', concatenated_dim0.shape)
print(concatenated_dim0)

concatenated_dim1 =  torch.cat((C1,C2),dim=1)
print('Concatenated along dimension 1:', concatenated_dim1.shape)
print(concatenated_dim1)

stacked =  torch.stack((C1,C2))
print('Stacked tensor shape:', stacked.shape)
print(stacked)
```

```
Concatenated along dimension 0: torch.Size([4, 3])
tensor([[0.7129, 0.5854, 0.0911],
        [0.8215, 0.0709, 0.4844],
        [0.8905, 0.4997, 0.4564],
        [0.2331, 0.0247, 0.0748]])
Concatenated along dimension 1: torch.Size([2, 6])
tensor([[0.7129, 0.5854, 0.0911, 0.8905, 0.4997, 0.4564],
        [0.8215, 0.0709, 0.4844, 0.2331, 0.0247, 0.0748]])
Stacked tensor shape: torch.Size([2, 2, 3])
tensor([[[0.7129, 0.5854, 0.0911],
         [0.8215, 0.0709, 0.4844]],

        [[0.8905, 0.4997, 0.4564],
         [0.2331, 0.0247, 0.0748]]])
```

Explain the diffrence between concatinating and stacking here

- Concatination adds two tensors along the dimensions that we specify , dim = 1 being along the columns and dim =0 along the rows
- Stacking adds two tensors by creating a new dimension

Q10: Advanced Indexing and Slicing (1 Point)

1. Given a tensor `D` with shape (6, 6), extract elements that are greater than 0.5.
2. Then, extract the second and fourth rows from `D`.
3. Finally, extract a sub-tensor from the top-left 3x3 block.

```
D = torch.rand(6, 6)
print(D)
print('Elements greater than 0.5:\n',  D[D>0.5])

second_fourth_rows =  D[1:5:2,:]
print('\nSecond and fourth rows:\n', second_fourth_rows)

top_left_3x3 =  D[0:3,3:7]
print('\nTop-left 3x3 block:\n ', top_left_3x3)
```

```
tensor([[0.8843, 0.4535, 0.1465, 0.0275, 0.8650, 0.6369],
        [0.7859, 0.9524, 0.6769, 0.1346, 0.9696, 0.6809],
        [0.9879, 0.9486, 0.7700, 0.7563, 0.0056, 0.0980],
        [0.1072, 0.2281, 0.9709, 0.0668, 0.2562, 0.2395],
        [0.6370, 0.9063, 0.2600, 0.6914, 0.9543, 0.7330],
        [0.9842, 0.0805, 0.0595, 0.4075, 0.1624, 0.3691]])
Elements greater than 0.5:
 tensor([0.8843, 0.8650, 0.6369, 0.7859, 0.9524, 0.6769, 0.9696, 0.6809, 0.9879,
        0.9486, 0.7700, 0.7563, 0.9709, 0.6370, 0.9063, 0.6914, 0.9543, 0.7330,
        0.9842])

Second and fourth rows:
 tensor([[0.7859, 0.9524, 0.6769, 0.1346, 0.9696, 0.6809],
        [0.1072, 0.2281, 0.9709, 0.0668, 0.2562, 0.2395]])

Top-left 3x3 block:
  tensor([[0.0275, 0.8650, 0.6369],
        [0.1346, 0.9696, 0.6809],
        [0.7563, 0.0056, 0.0980]])
```

Q11: Tensor Mathematical Operations (1 Point)

1. Create a tensor `G` with values from 0 to π in steps of π/4.
2. Compute and print the sine, cosine, and tangent logarithm and the exponential of `G`.

```
import math
```

```
G = torch.linspace(0,math.pi, steps=5)
print('G:', G)
print('Sine of G:',  torch.sin(G))
print('Cosine of G:',  torch.cos(G))
print('Tangent of G:', torch.tan(G))
print('Natural logarithm of G:',  torch.log(G))
print('Exponential of G:',  torch.exp(G))
```

```
G: tensor([0.0000, 0.7854, 1.5708, 2.3562, 3.1416])
Sine of G: tensor([ 0.0000e+00,  7.0711e-01,  1.0000e+00,  7.0711e-01, -8.7423e-08])
Cosine of G: tensor([ 1.0000e+00,  7.0711e-01, -4.3711e-08, -7.0711e-01, -1.0000e+00])
Tangent of G: tensor([ 0.0000e+00,  1.0000e+00, -2.2877e+07, -1.0000e+00,  8.7423e-08])
Natural logarithm of G: tensor([   -inf, -0.2416,  0.4516,  0.8570,  1.1447])
Exponential of G: tensor([ 1.0000,  2.1933,  4.8105, 10.5507, 23.1407])
```

Q12: **Tensor Reduction Operations (1 Point)**

1. Create a 3x2 tensor `H`.
2. Compute the sum of `H`. Print the result and shape after taking sun.
3. Then, perform the same operations along dimension 0 and dimension 1, printing the results and shapes.
4. What do you observe? How the shape changes?

```
H = torch.rand(3, 2)
print('H:', H, end = "\n\n")
print('Shape of original Tensor H', H.shape, end = "\n\n")

print('Sum of H:',  H.sum())
print('Shape after Sum of H:',  H.sum().shape,  end = "\n\n")

print('Sum of H along dimension 0:',  H.sum(axis=0))
print('Shape after sum of H along dimension 0:',  H.sum(axis=0).shape,  end = "\n\n")

print('Sum of H along dimension 1:', H.sum(axis=1))
print('Shape after sum of H along dimension 1:',  H.sum(axis=1).shape)
```

```
H: tensor([[0.8509, 0.1158],
        [0.9743, 0.9320],
        [0.0413, 0.3443]])

Shape of original Tensor H torch.Size([3, 2])

Sum of H: tensor(3.2587)
Shape after Sum of H: torch.Size([])

Sum of H along dimension 0: tensor([1.8665, 1.3922])
Shape after sum of H along dimension 0: torch.Size([2])

Sum of H along dimension 1: tensor([0.9667, 1.9063, 0.3857])
Shape after sum of H along dimension 1: torch.Size([3])
```

Provide your observations on shape changes here

- If we do the sum without mentioning any axis, We get a scalar, with) dimension
- As we mentioned sum with axis = 0, which means sum along the rows and we get a tensor with one dimension with 2 elements
- same with axis = 1 , sum along the columns, we get a tensor with one dimension with 3 elements

Q13: Working with Tensor Data Types (1 Point)

1. Create a tensor `I` of data type float with values `[1.0, 2.0, 3.0]`.
2. Convert `I` to data type int and print the result.
3. Explain in which scenarios it's necessary to be cautious about the data type of tensors.

```
# Solution for Q16
I =  torch.tensor([1.0,2.0,3.0])
print('I:', I)
I_int =  I.type(dtype=torch.int32)
print('I converted to int:', I_int)
```

```
I: tensor([1., 2., 3.])
I converted to int: tensor([1, 2, 3], dtype=torch.int32)
```

Your explanations here

- Created a tensor with values above mentioned, Converted the elements to mentioned dtype using type() function

Q14. Speedtest for vectorization 1.5 Points

Your goal is to measure the speed of linear algebra operations for different levels of vectorization.

1. Construct two matrices A and B with Gaussian random entries of size 1024×1024.
2. Compute $C = AB$ using matrix-matrix operations and report the time. (Hint: Use torch.mm)
3. Compute $C = AB$, treating A as a matrix but computing the result for each column of B one at a time. Report the time. (hint use torch.mv inside a for loop)
4. Compute $C = AB$, treating A and B as collections of vectors. Report the time. (Hint: use torch.dot inside nested for loop)

```
## Solution 1
torch.manual_seed(42) # dod not chnage this
A = torch.randn(1024,1024)
B = torch.randn(1024,1024)
```

```
## Solution 2
start=time.time()

C = torch.mm(A,B)
print("Matrix by matrix: " + str(time.time()-start) + " seconds")
```

```
Matrix by matrix: 0.04757523536682129 seconds
```

```
## Solution 3
C= torch.empty(1024,1024)
start = time.time()

for j in range(1024):
  C[:,j] = torch.mv(A,B[:,j])


print("Matrix by vector: " + str(time.time()-start) + " seconds")
```

```
Matrix by vector: 0.13890290260314941 seconds
```

```
## Solution 4
C= torch.empty(1024,1024)
start = time.time()

for i in range(1024):
  for j in range(1024):
    C[i,j] = torch.dot(A[i],B[j])
print("vector by vector: " + str(time.time()-start) + " seconds")
```

```
vector by vector: 14.959405899047852 seconds
```

Q15 : Redo Question 14 by using GPU - 1.5 Points

Using GPUs

How to use GPUs in Google Colab

In Google Colab -- Go to Runtime Tab at top -- select change runtime type -- for hardware accelartor choose GPU

```
# Check if GPU is availaible
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda:0
```

```
## Solution 1
torch.manual_seed(42)
A= torch.randn((1024, 1024),device=device)
B= torch.randn((1024, 1024),device=device)
```

```
## Solution 2
start=time.time()

C = torch.mm(A,B)

print("Matrix by matrix: " + str(time.time()-start) + " seconds")
```

```
Matrix by matrix: 0.11333036422729492 seconds
```

```
## Solution 3
C= torch.empty(1024,1024, device = device)
start = time.time()

for i in range(1024):
  C[:,i] = torch.mv(A,B[:,i])

print("Matrix by vector: " + str(time.time()-start) + " seconds")
```

```
Matrix by vector: 0.11850881576538086 seconds
```

```
## Solution 4
C= torch.empty(1024,1024, device = device)
start = time.time()

for i in range(1024):
  for j in range(1024):
    C[i,j]= torch.dot(A[i],B[j])

print("vector by vector: " + str(time.time()-start) + " seconds")
```

```
vector by vector: 30.659170627593994 seconds
```