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**

```
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[0,2] = 10
my_tensor[2,0] = 100
```

```
my_tensor.shape
```

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

```
my_tensor
```

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

```
x = x.view(-1,4).T
# x = x.view(-1,4).permute(1,0)
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 th
# Think about how you can select all columns for the desired row.
last_row = x[-1,:]
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
# 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
# Hint: Use stride slicing to extract the required subset of rows and columns
odd_valued_rows_columns = x[1::2, 1::2]
odd_valued_rows_columns
```

```
tensor([[7, 8]])
```

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

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

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

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

- 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
  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
  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])
```

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

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

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: 136932401851552
Memory address of A after in-place addition: 136932401851552
A after in-place addition: tensor([6, 7, 8])
Original memory address of B: 136932401852912
Memory address of B after out-of-place addition: 136932401852832
B after out-of-place addition: tensor([ 9, 10, 11])
```

Provide Explanation for above question here :

1. In-place operations modify the content of the initial tensor, while out-of-place operations produce a new tensor holding the altered data

produce a new tensor holding the altered data.

2. In the case of an in-place operation, the original tensor's memory location remains unaffected, as the modification occurs directly within it.
3. Conversely, out-of-place operations involve the creation of a new tensor containing the updated data, resulting in a different memory address compared to the original tensor.

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.randint(low=0, high=20, size=(3,1))
Y = torch.randint(low=0, high=20, size=(1,3))
print('Original shapes:', X.shape, Y.shape)
result = X + Y
print('Result:', result)
print('Result shape:', result.shape)
```

```
Original shapes: torch.Size([3, 1]) torch.Size([1, 3])
Result: tensor([[19, 30, 26],
        [14, 25, 21],
        [15, 26, 22]])
Result shape: torch.Size([3, 3])
```

Provide Explanation for above question here :

In this operation, broadcasting allows tensors X and Y to be added together, despite their differing shapes. Broadcasting expands the dimensions of each tensor to match the dimensions of the other tensor, facilitating element-wise addition. As a result, the final shape of the result tensor is (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.

```
torch.randint(low=0, high=10, size=(4,))
```

```
tensor([0, 2, 1, 2])
```

```
M1 = torch.randint(low=0, high=10, size=(3,5))
M2 = torch.randint(low=0, high=10, size=(5,2))
mat_multiplication =  M1 @ M2
print('Matrix multiplication result:', mat_multiplication)

V1 = torch.randint(low=0, high=10, size=(4,))
V2 = torch.randint(low=0, high=10, size=(4,))
dot_product =  torch.dot(V1, V2)
print('Dot product:', dot_product)
```

```
Matrix multiplication result: tensor([[ 76,  91],
        [ 68, 127],
        [110, 153]])
Dot product: tensor(44)
```

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)
T_view = T.view(3,8)
print('T_view shape:', T_view.shape)

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

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

```
T_view shape: torch.Size([3, 8])
T_reshape shape: torch.Size([4, 2, 3])
T_permute shape: torch.Size([4, 3, 2])
```

Provide Explanation for above question here : In PyTorch, `torch.view` is faster than `torch.reshape` for large tensors because it operates in-place, modifying the shape of the original tensor without creating a copy. This makes `torch.view` more memory-efficient and faster, especially when dealing with substantial amounts of data.

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 along 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)

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

stacked =  torch.stack((C1,C2), dim=0)
print('Stacked tensor shape:', stacked.shape)
```

```
Concatenated along dimension 0: torch.Size([4, 3])
Concatenated along dimension 1: torch.Size([2, 6])
Stacked tensor shape: torch.Size([2, 2, 3])
```

Explain the diffrence between concatinating and stacking here: The main difference between torch.cat and torch.stack lies in how they concatenate tensors:

torch.cat: Concatenates tensors along an existing dimension. The dimension along which concatenation occurs must have the same size in all tensors.

torch.stack: Stacks tensors along a new dimension. Introduces a new dimension and concatenates along that dimension. All tensors must have the same size along all existing dimensions.

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('Elements greater than 0.5:\n',  D[D>0.5])

second_fourth_rows =  D[[1,3][:]]
print('\nSecond and fourth rows:\n', second_fourth_rows)

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

```
Elements greater than 0.5:
 tensor([0.6010, 0.5505, 0.8873, 0.5304, 0.8724, 0.7596, 0.7690, 0.6234, 0.50
        0.8096, 0.6256, 0.9673, 0.7555, 0.5956, 0.5017, 0.5706, 0.6489, 0.643
        0.8858])

Second and fourth rows:
 tensor([[0.4315, 0.8724, 0.2084, 0.4635, 0.1819, 0.1901],
        [0.0099, 0.6256, 0.9673, 0.7555, 0.1982, 0.5956]])

Top-left 3x3 block:
  tensor([[0.6010, 0.5505, 0.8873],
        [0.4315, 0.8724, 0.2084],
        [0.7596, 0.3431, 0.7690]])
```

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 numpy as np
```

```
G = torch.linspace(0, np.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, -8.742
Cosine of G: tensor([ 1.0000e+00,  7.0711e-01, -4.3711e-08, -7.0711e-01, -1.0
Tangent of G: tensor([ 0.0000e+00,  1.0000e+00, -2.2877e+07, -1.0000e+00,  8.
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 =

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([[0.6582, 0.3452],
        [0.3311, 0.3333],
        [0.3929, 0.2525]])

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

Sum of H: tensor(2.3133)
Shape after Sum of H: torch.Size([])

Sum of H along dimension 0: tensor([1.3823, 0.9310])
Shape after sum of H along dimension 0: torch.Size([2])

Sum of H along dimension 1: tensor([1.0034, 0.6644, 0.6454])
Shape after sum of H along dimension 1: torch.Size([3])
```

Provide your observations on shape changes here

Observations:

- H.sum(): Summing all elements in tensor H yields a scalar value.
- H.sum(axis=0): Summing along dimension 0 collapses rows, resulting in a tensor with shape (2,), representing column sums.
- H.sum(axis=1): Summing along dimension 1 collapses columns, resulting in a tensor with shape (3,), representing row sums.

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., 2., 3.])
print('I:', I)
I_int =  I.int()
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

Caution about data types is crucial in scenarios where:

1. Precision matters: Using inadequate data types may lead to loss of information and inaccuracies in calculations involving decimals.
2. Compatibility with operations: Certain computations require specific data types for proper functionality. Using incompatible types can cause errors or unexpected outcomes.
3. Memory efficiency: Choosing suitable data types affects memory usage and computational efficiency. Using overly large types can result in excessive memory consumption and slower computations.

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: 0.033551931381225586 seconds
```

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

for i in range(B.shape[-1]):
  C[:,i] = torch.mv(A, B[:,i])

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

```
Matrix by vector: 0.27141404151916504 seconds
```

```
B.shape[-1]
```

```
1024
```

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

for i in range(A.shape[0]):
  for j in range(B.shape[-1]):
    C[i,j] = torch.dot(A[i,:], B[:,j])

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

```
vector by vector: 29.248013973236084 seconds
```

```
C.shape
```

```
torch.Size([1024, 1024])
```

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: 0.0008223056793212891 seconds
```

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

for i in range(B.shape[-1]):
  C[:,i] = torch.mv(A, B[:,i])

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

```
Matrix by vector: 0.05939173698425293 seconds
```

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

for i in range(A.shape[0]):
  for j in range(B.shape[-1]):
    C[i,j] = torch.dot(A[i,:], B[:,j])

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

```
vector by vector: 45.12138652801514 seconds
```

My idea: The function might be experiencing longer execution times on the GPU because the operation being performed is relatively simple. GPUs typically excel in accelerating more complex computations, and for simpler operations, the overhead of data transfer between CPU and GPU might outweigh the potential speedup.