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**

In [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.

In [2]:
```
my_tensor = torch.zeros(5, 3)
```

In [3]:
```
my_tensor.shape
```

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

In [4]:
```
my_tensor
```

Out[4]:
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

In [5]:
```
# 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 "r

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

In [6]:
```
my_tensor
```

Out[6]:
```
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]])
```

In [7]:
```
x=torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
```

In [8]:
```
x = x.view(6, 4).transpose(0,1)
x
```

Out[8]:
```
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

In [9]:
```
x = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 8, 10], [11, 12, 13, 14, 15]])
x
```

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

In [10]:
```
x.shape
```

Out[10]:
```
torch.Size([3, 5])
```

In [11]:
```
# Student Task: Retrieve the last row of the tensor 'x'
# Hint: Negative indexing can help you select rows or columns counting from the
# Think about how you can select all columns for the desired row.
last_row = x[-1, :]
last_row
```

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

In [12]:
```
# 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
```

Out[12]:
```
tensor([ 4,  8, 14])
```

In [13]:
```
# 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
```

Out[13]:
```
tensor([[ 1,  2],
        [ 6,  7],
        [11, 12]])
```

In [14]:
```
# Student Task: Retrieve the rows and columns with odd-indexed positions from t
# Hint: Use stride slicing to extract the required subset of rows and columns w
odd_valued_rows_columns = x[::2, ::2]
odd_valued_rows_columns
```

Out[14]:
```
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.

In [15]:
```
# Given Data
x = [[ 3,  60,  100, -100],
     [ 2,  20,  600, -600],
     [-5,  50,  900, -900]]
```

In [16]:
```
# Convert to PyTorch Tensor and set to float
X = torch.tensor(x)
X= X.float()
```

In [17]:
```
# Print shape and data type for verification
print(X.shape)
print(X.dtype)
```

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

In [18]:
```
# Compute and display the mean and standard deviation of each column for refere
X.mean(axis = 0)
X.std(axis = 0)
```

Out[18]:
```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

In [19]:
```
X.std(axis = 0)
```

Out[19]:
```
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.

In [20]:
```
def normalize_matrix(x):
  # Calculate the mean along each column (think carefully , you will take mean
  mean = torch.mean(x, dim=0)

  # Calculate the standard deviation along each column
  std = torch.std(x, dim=0)

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

  return y  # Return the normalized matrix
```

In [21]:
```
Z = normalize_matrix(X)
Z
```

Out[21]:
```
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]])
```

In [22]:
```
Z.mean(axis = 0)
```

Out[22]:
```
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

In [23]:
```
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: 5434774480
Memory address of A after in-place addition: 5434774480
A after in-place addition: tensor([6, 7, 8])
Original memory address of B: 5434774288
Memory address of B after out-of-place addition: 5434773040
B after out-of-place addition: tensor([ 9, 10, 11])
```

Provide Explanation for above question here :

A is a in place operation. This ensures memory address of the tensor remains the same and helps with memory allocation.

B is being created again and assigned the same variable name. Therefore, each time B is created it is alloted a new address.

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.

In [24]:
```
X = torch.rand((3, 1))
Y = torch.rand((1, 3))
print('Original shapes:', X.shape, Y.shape)
print(X)
print(Y)
result = X + Y
print('Result:', result)
print('Result shape:', result.shape)
```

```
Original shapes: torch.Size([3, 1]) torch.Size([1, 3])
tensor([[0.0992],
        [0.1627],
        [0.6817]])
tensor([[0.3329, 0.4364, 0.1186]])
Result: tensor([[0.4321, 0.5356, 0.2178],
        [0.4956, 0.5991, 0.2813],
        [1.0146, 1.1180, 0.8002]])
Result shape: torch.Size([3, 3])
```

Provide Explanation for above question here :

Broadcasting enables the tensors X and Y to be of same shape and it used to perform the Tensor addition

- PyTorch identifies compatible dimensions (here, the first dimension of X and the second dimension of Y have a size of 1).

- It virtually expands those dimensions to match each other, creating intermediate views of X and Y as if they both had a shape of (3, 3).
- This allows element-wise addition to proceed without explicit reshaping.

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.

In [25]:
```
M1 = torch.ones(3,2)
M2 = torch.ones(2,3)
mat_multiplication =  torch.mm(M1,M2)
print('Matrix multiplication result:', mat_multiplication)

V1 = torch.ones(3,)
V2 =  torch.ones(3,)
dot_product =  torch.dot(V1, V2)
print('Dot product:', dot_product)
```

```
Matrix multiplication result: tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
Dot product: tensor(3.)
```

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

In [26]:
```
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 :

View can be only used for contiguous tensor in contrary to reshape.

If you just want to reshape tensors, use torch.reshape. If you're also concerned about memory usage and want to ensure that the two tensors share the same data, use torch.view.

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.

In [27]:

```
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

Concatinating combines tensors along an existing dimnesion, whereas stacking adds a dimension and adds the tensors along that 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.

In [28]:

```
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([0.8305, 0.5064, 0.9334, 0.8831, 0.8864, 0.8975, 0.5566, 0.9483, 0.876
6,
        0.5091, 0.9582, 0.6166, 0.8757, 0.9742, 0.7940, 0.6839, 0.8771, 0.777
1])

Second and fourth rows:
 tensor([[0.1285, 0.0165, 0.3430, 0.0222, 0.3897, 0.3624],
        [0.8766, 0.5091, 0.9582, 0.6166, 0.8757, 0.1016]])

Top-left 3x3 block:
  tensor([[0.2052, 0.8305, 0.5064],
        [0.1285, 0.0165, 0.3430],
        [0.1820, 0.8831, 0.8864]])
```

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`.

In [29]:
```
import numpy as np
G = torch.arange(0, torch.tensor(np.pi), step=torch.tensor(np.pi / 4))

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])
Sine of G: tensor([0.0000, 0.7071, 1.0000, 0.7071])
Cosine of G: tensor([ 1.0000e+00,  7.0711e-01, -4.3711e-08, -7.0711e-01])
Tangent of G: tensor([ 0.0000e+00,  1.0000e+00, -2.2877e+07, -1.0000e+00])
Natural logarithm of G: tensor([   -inf, -0.2416,  0.4516,  0.8570])
Exponential of G: tensor([ 1.0000,  2.1933,  4.8105, 10.5507])
```

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?

In [30]:

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

print('Sum of H along dimension 0:',  torch.sum(H, dim = 0))
print('Shape after sum of H along dimension 0:',  torch.sum(H, dim = 0).shape,

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

```
H: tensor([[0.2863, 0.8445],
        [0.3185, 0.5585],
        [0.8606, 0.2818]])

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

Sum of H: tensor(3.1502)
Shape after Sum of H: torch.Size([])

Sum of H along dimension 0: tensor([1.4654, 1.6848])
Shape after sum of H along dimension 0: torch.Size([2])

Sum of H along dimension 1: tensor([1.1308, 0.8770, 1.1424])
Shape after sum of H along dimension 1: torch.Size([3])
```

Provide your observations on shape changes here

1. Is the sum of all the scalars in the Tensor H, therefore it is a scalar.

2. Is the sum along dimension 0 (rows), therefore it is a scalar of dimension (2,).

3. Is the sum along dimension 1 (columns), therefore it is a scalar of dimension (3,).

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.

In [31]:

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

Memory usage, precision loss and gradient computation are the reasons to be cautiuos.

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)

In [32]:
```
## Solution 1
torch.manual_seed(42) # dod not chnage this
A = torch.randn(1024, 1024)
B = torch.randn(1024, 1024)
```

In [33]:
```
## Solution 2
start=time.time()

C = torch.mm(A, B)

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

```
Matrix by matrix: 0.0035250186920166016 seconds
```

In [34]:
```
## Solution 3
C= torch.empty(1024,1024)
start = time.time()

C = torch.zeros(1024, 1024)
for i in range(B.size(1)):
    C[:, i] = torch.mv(A, B[:, i])

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

```
Matrix by vector: 0.06011795997619629 seconds
```

In [35]:
```
## Solution 4
C= torch.empty(1024,1024)
start = time.time()

C = torch.zeros(1024, 1024)
for i in range(A.size(1)):
    for j in range(B.size(1)):
        C[i, j] = torch.dot(A[:, i], B[:, j])

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

```
vector by vector: 11.896167039871216 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

In [36]:
```
# Check if GPU is availaible
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cpu
```

In [37]:
```
## Solution 1
torch.manual_seed(42)
A= torch.randn((1024, 1024),device=device)
B= torch.randn((1024, 1024),device=device)
```

In [38]:
```
print(A)
print(B)
```

```
tensor([[ 1.9269,  1.4873,  0.9007,  ...,  1.1085,  0.5544,  1.5818],
        [-1.2248,  0.9629, -1.5785,  ..., -0.0334, -0.8276, -0.3524],
        [-0.6002, -0.0580,  0.2975,  ..., -2.3891,  0.7178, -1.5831],
        ...,
        [-0.1277,  0.2762,  0.9963,  ...,  0.5773,  0.4467,  1.6042],
        [-1.2248, -0.0074, -1.2422,  ..., -0.0288,  1.1666, -0.2757],
        [-1.3839, -0.2788, -0.7896,  ..., -1.0240, -0.4006,  0.2859]])
tensor([[-0.6495,  0.6962,  2.4458,  ...,  0.8676, -0.7841,  0.4688],
        [-1.8024,  1.1675, -1.0891,  ...,  1.8599, -0.6542, -1.1773],
        [-0.0060, -1.6660,  0.4924,  ..., -0.1280, -0.6696,  0.8153],
        ...,
        [ 0.8675,  2.4424, -1.4791,  ..., -1.4814,  1.6032, -0.7838],
        [ 0.8459,  1.0714, -0.3842,  ..., -1.2028, -0.5702,  0.1948],
        [-0.3129,  0.3797, -1.1664,  ..., -0.0991, -2.6005,  0.1010]])
```

In [39]:
```
## Solution 2
start=time.time()

C = torch.mm(A, B)

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

```
Matrix by matrix: 0.003468036651611328 seconds
```

In [40]:
```
## Solution 3
C= torch.empty(1024,1024, device = device)
start = time.time()

C = torch.zeros(1024, 1024)
```

```python
for i in range(B.size(1)):
    C[:, i] = torch.mv(A, B[:, i])

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

```
Matrix by vector: 0.05757904052734375 seconds
```

In [41]:

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

C = torch.zeros(1024, 1024)
for i in range(A.size(1)):
    for j in range(B.size(1)):
        C[i, j] = torch.dot(A[:, i], B[:, j])

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

```
vector by vector: 12.009110689163208 seconds
```