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.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] = 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, 18, 19, 20, 21, 22, 23])
```

```
x = x.view(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[-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 '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])
```

```
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 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: 139199110429280
Memory address of A after in-place addition: 139199110429280
A after in-place addition: tensor([6, 7, 8])
Original memory address of B: 139199129650416
Memory address of B after out-of-place addition: 139199108657968
B after out-of-place addition: tensor([ 9, 10, 11])
```

Provide Explanation for above question here : For tensor A, the memory address remains the same before and after the in-place addition (A.add_(5)). This indicates that the operation modified the original tensor A in memory, resulting in the values [6, 7, 8].

For tensor B, the memory address changes after the out-of-place addition (B = B + 5). This indicates that a new tensor was created for the result, and the original B tensor was reassigned to this new tensor, resulting in the values [9, 10, 11].

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.arange(3).reshape(3,1)
Y = X.T
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([[0, 1, 2],
        [1, 2, 3],
        [2, 3, 4]])
Result shape: torch.Size([3, 3])
```

Provide Explanation for above question here : This operation involves broadcasting because the shapes of X and Y are different. Broadcasting rules are applied to allow the addition to take place.

Broadcasting Rule 1: If the tensors have a different number of dimensions, pad the smaller shape with ones on the left side until they have the same number of dimensions. In this case, Y is padded to shape (1, 3).

Broadcasting Rule 2: Compare the sizes of the dimensions element-wise. Two dimensions are compatible when either they are equal or one of them is 1. In this case, the dimensions are compatible because they are either equal or one of them is 1.

Broadcasting Rule 3: If the sizes are not equal in any dimension, and none of them is 1, then the operation is not possible.

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.arange(0, 12, dtype=float).reshape(2, 6)
M2 = torch.ones(6, 2, dtype=float)
mat_multiplication =  torch.mm(M1, M2)
print('Matrix multiplication result:', mat_multiplication)

V1 = torch.tensor([1,3,5,7])
V2 = torch.tensor([2,4,6,8])
dot_product = torch.dot(V1,V2)
print('Dot product:', dot_product)
```

```
Matrix multiplication result: tensor([[15., 15.],
        [51., 51.]], dtype=torch.float64)
Dot product: tensor(100)
```

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 : View will not create a copy and will allow us to perform fast and memory efficient computations whereas reshape may or may not share the same memory.

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)

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

stacked = torch.stack((C1,C2))
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 torch.cat combines tensors along an existing dimension, while torch.stack creates a new dimension to stack tensors. The choice between them depends on whether you want to preserve the existing dimensions or create a new one in the result.

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([0.7813, 0.7276, 0.6424, 0.5090, 0.8774, 0.8632, 0.5983, 0.6484, 0.5029,
        0.6347, 0.5238, 0.8120, 0.8766, 0.5487, 0.8726, 0.9057, 0.7080, 0.9998])

Second and fourth rows:
 tensor([[0.6424, 0.5090, 0.3347, 0.3164, 0.4714, 0.4695],
        [0.5029, 0.6347, 0.1354, 0.5238, 0.8120, 0.1768]])

Top-left 3x3 block:
  tensor([[0.1153, 0.2992, 0.7813],
        [0.6424, 0.5090, 0.3347],
        [0.8774, 0.8632, 0.0095]])
```

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.arange(0, math.pi + math.pi/4, math.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, 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(dim=0))
print('Shape after sum of H along dimension 0:',  H.sum(dim=0).shape,  end = "\n\n")

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

```
H: tensor([[0.3559, 0.6151],
        [0.8787, 0.6388],
        [0.9880, 0.8760]])

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

Sum of H: tensor(4.3525)
Shape after Sum of H: torch.Size([])

Sum of H along dimension 0: tensor([2.2226, 2.1299])
Shape after sum of H along dimension 0: torch.Size([2])

Sum of H along dimension 1: tensor([0.9710, 1.5175, 1.8641])
Shape after sum of H along dimension 1: torch.Size([3])
```

Provide your observations on shape changes here Summing all elements: If you sum all elements in the tensor without specifying a dimension, the tensor collapses into a single scalar value, effectively removing all dimensions. This is why the shape becomes an empty tuple, indicating a scalar rather than a tensor.

Summing along a specific dimension: When you sum along a specific dimension of a tensor, you collapse that dimension, reducing the overall dimensionality of the tensor by one. The size of the remaining dimensions is preserved in the shape of the resulting tensor.

Summing along dimension 0 (e.g., H.sum(dim=0)) means you are summing across rows for each column, resulting in a tensor that represents the column-wise sum. This collapses the row dimension, leaving you with a 1D tensor whose length matches the number of columns in the original tensor. Summing along dimension 1 (e.g., H.sum(dim=1)) means you are summing across columns for each row, resulting in a tensor that represents the row-wise sum. This collapses the column dimension, leaving you with a 1D tensor whose length matches the number of rows in the original tensor.

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 =  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 Loss of Precision: Converting from float to int results in the loss of decimal part, which can be significant in calculations where precision matters, such as in scientific computations.

Overflow and Underflow: When converting to a data type with a smaller range, values might exceed the range of the target type (overflow) or be too small to represent (underflow), leading to unexpected results or errors.

Computational Requirements: Certain operations and neural network layers require inputs of specific data types. Using the wrong type can lead to runtime errors or inefficient computations.

Memory Usage: Data types with larger sizes (like float64) consume more memory compared to smaller types (like int8). Converting to a smaller data type can save memory, but at the cost of losing information or range.

Model Accuracy: In machine learning and deep learning, the precision of weights, biases, and inputs can significantly affect the accuracy of the model. Lower precision might speed up computations but degrade model performance.

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: 0.045966148376464844 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: 0.31170654296875 seconds
```

```
## 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: 35.58580207824707 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
import torch
import time
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: 0.10692691802978516 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: 0.04037213325500488 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: 44.12438249588013 seconds
```