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**

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
pip install torch
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-pacl
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/pytl
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-pacl
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packa
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.wl
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch)
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl.metad
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl.meta
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl.meta
Collecting nvidia-curand-cu12==10.3.2.106 (from torch)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl.me
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch)
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch)
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl
Collecting nvidia-nccl-cu12==2.20.5 (from torch)
  Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl.metad
Collecting nvidia-nvtx-cu12==12.1.105 (from torch)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.metada
```

```
Requirement already satisfied: triton==2.3.1 in /usr/local/lib/python3.10/dis
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torc
  Using cached nvidia_nvjitlink_cu12-12.6.20-py3-none-manylinux2014_x86_64.wh
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.1
Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.
Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (
Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (
Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7
Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.
Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56
Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl (
Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl (
Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl (176.2
Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
Using cached nvidia_nvjitlink_cu12-12.6.20-py3-none-manylinux2014_x86_64.whl
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidi
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.
```

```
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)) # CODE HERE


# Modify the tensor to set element (0, 2) to 10 and element (2, 0) to 100
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_
```

```
# CODE HERE

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,


# Reshape the tensor into a 6x4 matrix
x = x.view(6, 4)

print(x)
```

```
    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]])
```

```
# Transpose the matrix
x = x.t()

print(x)
```

```
    tensor([[ 0,  4,  8, 12, 16, 20],
            [ 1,  5,  9, 13, 17, 21]
```

```
[ 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 e
# Think about how you can select all columns for the desired row.
last_row = x[-1, :] # CODE HERE
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] # CODE HERE
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]# CODE HERE
first_3_rows_2_columns
```

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

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

```
# Student Task: Retrieve the rows and columns with odd-indexed positions from the
# Hint: Use stride slicing to extract the required subset of rows and columns wit
odd_valued_rows_columns = x[0::2, 0::2]# CODE HERE
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 referenc
X.mean(axis = 0)
```

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

Start coding or generate with AI.

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

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

Your task starts here

- 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 (axis 0)
    mean = x.mean(dim=0)

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

```
# Step 1: Create a tensor `A` with values [1, 2, 3]
```

```
# Step 1: Create a tensor `A` with values [1, 2, 3]
A = torch.tensor([1, 2, 3])
print('Original memory address of A:', id(A))

# Step 2: Perform in-place addition of 5 to tensor `A`
A.add_(5)
print('Memory address of A after in-place addition:', id(A))
print('A after in-place addition:', A)

# Step 3: Create a tensor `B` with values [4, 5, 6]
B = torch.tensor([4, 5, 6])
print('Original memory address of B:', id(B))

# Step 4: Perform out-of-place addition of 5 to tensor `B`
B = B + 5  # This creates a new tensor and assigns it to `B`
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: 132363084169600
Memory address of A after in-place addition: 132363084169600
A after in-place addition: tensor([6, 7, 8])
Original memory address of B: 132363084169840
Memory address of B after out-of-place addition: 132363084169760
B after out-of-place addition: tensor([ 9, 10, 11])
```

Provide Explanation for above question here :

A. when you use in-place addition A.add_(5) , the operation changes the tensor A directly, therefore the memory address of A remains the same after the operation, as there is no need to create a new tensor

B. when you perform an out-of-place operation, like B+5, a new tensor is created and the memory address of 'B' changes after the operation. For this operation, a new tensor is created and the results are stored back to the original tensor B

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.

```
# Create tensor X with shape (3, 1)
X = torch.tensor([[1], [2], [3]])

# Create tensor Y with shape (1, 3)
Y = torch.tensor([[4, 5, 6]])

# Print the original shapes
print('Original shapes:', X.shape, Y.shape)

# Perform the addition
result = X + Y

# Print the result and its shape
print('Result:', result)
print('Result shape:', result.shape)
```

```
Original shapes: torch.Size([3, 1]) torch.Size([1, 3])
Result: tensor([[5, 6, 7],
        [6, 7, 8],
        [7, 8, 9]])
Result shape: torch.Size([3, 3])
```

Provide Explanation for above question here : Initially, I created the tensors with different dimensions. Broadcasting allows for operations on tensors of different shapes by expanding one or both tensors to a compatible shape without copying data. In this case, X with shape (3, 1) was broadcasted to (3, 3) by repeating its single column across the other columns. Similarly, Y with shape (1, 3) was broadcasted to (3, 3) by repeating its single row across the other rows. The resulting tensor Z has a shape of (3, 3) and contains the element-wise sum of the broadcasted tensors.

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.tensor([[1, 2], [3, 4]])

M2 = torch.tensor([[5, 6], [7, 8]])

# Perform matrix multiplication
mat_multiplication = torch.mm(M1, M2)
```

```
print('Matrix multiplication result:', mat_multiplication)

# Step 2: Create two vectors V1 and V2
V1 = torch.tensor([1, 2, 3])
V2 = torch.tensor([4, 5, 6])

# Compute the dot product of V1 and V2
dot_product = torch.dot(V1, V2)
print('Dot product:', dot_product)
```

```
Matrix multiplication result: tensor([[19, 22],
        [43, 50]])
Dot product: tensor(32)
```

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)

# 1. Reshape using `view` to (3, 8)
T_view = T.view(3, 8)
print('T_view shape:', T_view.shape)

# 2. Reshape using `reshape` to (4, 2, 3)
T_reshape = T.reshape(4, 2, 3)
print('T_reshape shape:', T_reshape.shape)

# 3. Transpose the first and last dimensions using `permute`
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: Used to reshape the tensor T from (2, 3, 4) to (3, 8) by changing its shape without altering the underlying data layout.

reshape: Reshaped the tensor T from (2, 3, 4) to (4, 2, 3) and handled the tensor's non-contiguous memory by creating a new tensor with the desired shape.

permute: Reordered the dimensions of T from (2, 3, 4) to (4, 3, 2), changing the order of the dimensions without altering the tensor's 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 alomng dimension 0 and print the shape of the resulting tensor.
3. What is the difference between stacking and concatinating.

```
# Create tensors C1 and C2 with shape (2, 3)
C1 = torch.rand(2, 3)
C2 = torch.rand(2, 3)

# 1. Concatenate along dimension 0
concatenated_dim0 = torch.cat((C1, C2), dim=0)
print('Concatenated along dimension 0:', concatenated_dim0.shape)

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

# 3. Stack along dimension 0
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

Concatenation along Dimension 0: torch.cat((C1, C2), dim=0) This combines C1 and C2 along the first dimension (rows), so the result is a single tensor with more rows but the same number of columns. Resulting Shape: (4, 3) — C1 and C2 are stacked on top of each other.

Concatenation along Dimension 1: torch.cat((C1, C2), dim=1) This combines C1 and C2 along the second dimension (columns), so the result is a single tensor with more columns but the

same number of rows. Resulting Shape: (2, 6) — C1 and C2 are placed side by side.

Stacking along Dimension 0: torch.stack((C1, C2), dim=0) This creates a new dimension and stacks C1 and C2 along this new dimension. Each tensor becomes a separate "slice" along this new dimension. Resulting Shape: (2, 2, 3) — The new dimension (0) holds two layers (one for each of C1 and C2), each of shape (2, 3).

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)
# 1. Extract elements greater than 0.5
elements_greater_than_0_5 = D[D > 0.5]
print('Elements greater than 0.5:\n', elements_greater_than_0_5)

# 2. Extract the second and fourth rows
second_fourth_rows = D[[1, 3], :]
print('\nSecond and fourth rows:\n', second_fourth_rows)

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

```
Elements greater than 0.5:
 tensor([0.8292, 0.6681, 0.7118, 0.9073, 0.5648, 0.5332, 0.7644, 0.8222, 0.59
        0.7425, 0.9295, 0.5248, 0.7486])

Second and fourth rows:
 tensor([[0.0522, 0.4350, 0.1069, 0.5648, 0.1129, 0.1137],
        [0.8222, 0.0661, 0.5956, 0.3014, 0.1175, 0.7425]])

Top-left 3x3 block:
 tensor([[0.8292, 0.6681, 0.7118],
        [0.0522, 0.4350, 0.1069],
        [0.1725, 0.1402, 0.5332]])
```

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`.

```
G = torch.arange(0,torch.pi+torch.pi/4,step =torch.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[1:]))
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([-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?

```
# 1. Create a 3x2 tensor H
H = torch.rand(3, 2)
print('H:', H, end="\n\n")
print('Shape of original Tensor H:', H.shape, end="\n\n")

# 2. Compute the sum of H
sum_H = torch.sum(H)
print('Sum of H:', sum_H)
print('Shape after Sum of H:', sum_H.shape, end="\n\n")
# 3. Compute the sum along dimension 0
sum_H_dim0 = torch.sum(H, dim=0)
print('Sum of H along dimension 0:', sum_H_dim0)
print('Shape after sum of H along dimension 0:', sum_H_dim0.shape, end="\n\n")

# 4. Compute the sum along dimension 1
sum_H_dim1 = torch.sum(H, dim=1)
print('Sum of H along dimension 1:', sum_H_dim1)
print('Shape after sum of H along dimension 1:', sum_H_dim1.shape)
```

```
H: tensor([[0.4951, 0.3167],
        [0.0548, 0.2115],
        [0.1740, 0.8918]])
```

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

Sum of H: tensor(2.1440)
Shape after Sum of H: torch.Size([])

Sum of H along dimension 0: tensor([0.7240, 1.4200])
Shape after sum of H along dimension 0: torch.Size([2])

Sum of H along dimension 1: tensor([0.8118, 0.2663, 1.0659])
Shape after sum of H along dimension 1: torch.Size([3])
```

Provide your observations on shape changes here

In tensor reduction operations, summing all elements of a tensor results in a scalar with no dimensions, collapsing all axes into a single number. Summing along a specific dimension reduces that dimension while preserving others; for example, summing along dimension 0 of a 3x2 tensor results in a 1D tensor of shape (2,), representing the sum of each column, while summing along dimension 1 yields a 1D tensor of shape (3,), representing the sum of each row. Thus, summing across dimensions decreases the tensor's rank by collapsing the summed dimension while retaining the shape of the remaining dimensions.

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], dtype=torch.float) # CODE HERE
print('I:', I)
I_int = I.int()  #I.to(dtype=torch.int) # CODE HERE
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 When working with tensor data types, it's crucial to be cautious about conversions between types due to potential precision loss and compatibility issues. Converting from float to int truncates decimal values, which can lead to a loss of precision that might affect numerical accuracy in computations. also, different data types also impact memory usage and performance, with smaller types potentially reducing memory consumption but possibly sacrificing precision

sacrificing precision

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)

```
import time
##Solution 1

torch.manual_seed(42)

# 1. Construct two matrices A and B with Gaussian random entries of size 1024x102
A = torch.randn(1024, 1024)
B = torch.randn(1024, 1024)

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

```
Matrix by matrix: 0.07712960243225098 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: 0.2900547981262207 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: 35.57971477508545 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

```
import torch
import time
# 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) # code here

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

```
Matrix by matrix: 0.1557598114013672 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])# code here
```

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

```
Matrix by vector: 0.15434908866882324 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])# code here

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

```
vector by vector: 57.57485508918762 seconds
```

Experimenting with GPU and CPU options

```
#################################################################
### Turning off GPU to show the time difference On CPU and that I was able to make
```

```
Start coding or generate with AI.
```

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

```
cpu
```

```
## 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) # code here

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

```
Matrix by matrix: 0.40303468704223633 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])# code here

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

```
Matrix by vector: 2.328982353210449 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])# code here

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

```
vector by vector: 35.6391077041626 seconds
```

Start coding or generate with AI.