Tensors in Pytorch

By Md. Rishat Talukder

Pytorch is the successor of Torch which is an open-source machine learning library based on the Torch library. It is used for applications such as natural language processing(LLM), computer vision, audio processing, etc. It is primarily developed by Facebook's AI Research lab (FAIR).

Pytorch is a deep learning framework that provides a flexible and dynamic computational graph. It is a Python-based scientific computing package that uses the power of GPUs. It is also one of the preferred deep learning research platforms built to provide flexibility as a deep learning research platform.

If you have prior knowledge of Deep Learning and Neural Networks you should have heard the name of Tensorflow which is another popular deep learning framework. The main difference between Pytorch and Tensorflow is that Pytorch uses dynamic computation graphs while Tensorflow uses static computation graphs.

But both are widely populer in their own way. In this article series, we will learn about Pytorch and how to use it for Deep Learning.

I'm Md. Rishat Talukder and I will be your instructor in this article series. I'm a softwere/backend developer and AI enthusiast. I have a good amount of experience in Deep Learning and Neural Networks. I have worked on various projects using Pytorch and Tensorflow. I'm also a content creator and I have a Youtube channel where I make coding tutorials and share my knowledge with others. You can follow me on my social media platforms to stay updated with my latest works.

- Youtube
- LinkedIn
- Github

I'll make part by part small articles on Deep Learning with Pytorch where I'll cover all the things I learned and experienced in my Deep Learning journey. So, let's get started with our first part.

Series Prerequisites

- Basic knowledge of Python
- Basic knowledge of Numpy, Pandas, and Matplotlib.
- Basic knowledge of Machine Learning
- Basic knowledge of Deep Learning and Neural Networks

What are tensors?

tensorss are the building blocks of a deep learning model. They are the mathemetical representation of the data.

Importing the libraries

```
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

torch.__version__
```

```
'2.3.0+rocm6.0'
```

We have imported the tools now time to create some tensors.

Scaler

Scalers are single values, like a single integer value. This is called a non-dimentional tensor or `0-dimensional` tensor.

```
scaler = torch.tensor(420)
scaler
```

```
tensor(420)
```

```
print(f"dimensions: {scaler.ndim}")
```

```
dimensions: 0
```

Vector

`vectors` are arrays that has only a length and item on every index. We can say the vector is a 1-demensional array.

```
vector = torch.tensor([1, 2, 3, 4, 5])
```

```
print(f"shape: {vector.shape}")
```

```
shape: torch.Size([5])
```

```
print(f"dimension: {vector.ndim}")
```

```
dimension: 1
```

```
print(f"data type: {vector.dtype}")
```

```
data type: torch.int64
```

Matrix

I think you get the idea behind. It 0-dimensional means it's a scaler, 1-dimension means it's a vector. SO, `2-dimenssional` would be a `Matrix`.

```
MATRIX = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
print(f"shape: {MATRIX.shape}")
```

```
shape: torch.Size([3, 3])
```

```
print(f"dimansions: {MATRIX.ndim}")
```

```
dimansions: 2
```

Tensors

Finally anything above 2-dimensional is called a `tensor` itself.

> This is just a naming convention. Pytorch arrays are called tensors, so it does not matter how many dimansions there is Every data representation in pytorch is a tensor.

```
TENSORS = torch.tensor(
    [[[1, 2, 3], [4, 5, 6], [7, 8, 9]]]
)

TENSORS
```

```
tensor([[[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]])
```

```
TENSORS.shape
```

```
torch.Size([1, 3, 3])
```

```
TENSORS.ndim
```

```
3
```

> If you look at the examples I have given the MATRIX and TENSOR are capitalized. This is another naming convention where any tensor with higher dimensions then one should be capitalized. As a Matrix has 2-dimensions and a tensor has 3-dimensions Thats why the variable name is capitalized.

All of the above(`scaler`, `vector`, `matrix`, `tensors`) are tensors.

We call a 0-dimensional tensor a `scaler`, a 1-dimensional tensor a `vector`, a 2-dimensional tensor a `matrix`, and a tensor with more than two dimensions simply a `tensor`.

SOOOOOOOOOOOOOOOOOO

Simply Tensors Are N-Dimensional Arrays With Superpowers.

Random Tensors

Random tensors are a big part in `Pytorch`. Because when a `neural network` starts to learns it starts with tensors full of random numbers then they adjust those random number to better represent the data.

starts with random numbers -> look at the data -> update -> repeat step 2 and 3.

Let's make a random number tensor.

```
random_vector = torch.rand(10)
random_vector
```

```
tensor([0.1248, 0.4535, 0.6411, 0.7467, 0.7434, 0.5068, 0.9361, 0.8287,
0.1771,
        0.5501])
```

There just 1 command you have a `vector` or a 1-dimensional tensor. Not only that you can make n-dimansional tensors in an instant.

```
random_MATRIX = torch.rand(size=(4,5))
random_MATRIX,
```

```
(tensor([[0.6136, 0.1042, 0.0994, 0.1838, 0.0607],
         [0.6509, 0.2393, 0.0012, 0.4856, 0.0250],
         [0.8643, 0.4529, 0.9640, 0.0993, 0.9601],
         [0.9255, 0.4487, 0.0633, 0.2613, 0.1219]]),)
```

```
random_TENSOR = torch.rand(size=(2,3,4))
random_TENSOR
```

```
tensor([[[0.3125, 0.6634, 0.3643, 0.8087],
         [0.9163, 0.7944, 0.4506, 0.8632],
         [0.7823, 0.6702, 0.3306, 0.9175]],

        [[0.5322, 0.5879, 0.6275, 0.1062],
         [0.0189, 0.4362, 0.3783, 0.4945],
         [0.0956, 0.4862, 0.4486, 0.3104]]])
```

Now, I won't go deep into this random stuff because most of the functions here is the same as the numpy random library. So, if you know numpy random you should know how to apply thses random functionalties in data processing and other cool stuff.

There you go...

Tensors is a multidimensional array that is used to make a neural network and we can simply say "A Tensor is a Multidimensional Numpy Array with Superpowers", and we will learn more about it as we keep on making thing using the Pytorch library.

Ones & Zeros

There are other nice and easy things that can be done with `tensors`. For examples: Making a Tensor of all `zeros` or `Ones`.

Now before going into `how we can get a tensor full of Zeros and Ones with Pytorch`, we should try to make a it our selves.

Now if you have experienece with `numpy` then you should how numpy expression works.

> We can add, substract, multiply or divide a multidimensional array by a single number like this.

```
import numpy as np


normal_array = np.array([[1,2,3],
                         [2,3,4],
                         [3,4,5]])

normal_array
```

```
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```

Here's a normal numpy array. Now we can do any operations on the array with a single element which will be then applied to the whole 2-dimensional array we just made.

```
normal_array/2
```

```
array([[0.5, 1. , 1.5],
       [1. , 1.5, 2. ],
       [1.5, 2. , 2.5]])
```

AAAANNNNND there you go, you can clearly see that the numpy array changed and the whole array is affected.

Now do you remember what I said about Tensors???

> A Tensor is a multidimensional array with SuperPowers.

So, We can so all the things we could do with a numpy array and much much more with a Tensor.

In that case we should also have the ability to do mathemetical oprations to a Tensor like we can do to a numpy array.

```
normal_tensor = torch.rand(size=(3,4))
normal_tensor
```

```
tensor([[0.1082, 0.5997, 0.8875, 0.7790],
        [0.9564, 0.3755, 0.3976, 0.9715],
        [0.7895, 0.4885, 0.4484, 0.1327]])
```

Here we made a 2-dimensional 3 by 4 Tensor and now let's see what happens if we divide the tensor with `0`.

```
normal_tensor*0
```

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

AAAAAANNNNNNDDDD there you go. We have a 3 by 4, 2-dimensional `tensor with all zeros`.

Now how can we do the same if we want a tensor with all `Ones`????

This is your homework. Try to figure it out yourself...

🗨 Make a 3-dimensional tensor without using the built in `torch` function.

Now let's see how we can make a multidimensional tensor full of `zeros and Ones`.

In pyTorch there are `Two functions` You can use to Make a zeors or ones tensor.

```
torch.zeros(size=(3,4))
```

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

```
torch.ones(size=(3,4))
```

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

```
torch.zeros(size=(3,4,5))
```

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

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]]])
```

```
torch.ones(size=(3,4,5))
```

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

        [[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]],
```

```
        [[1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1.]]])
```

This is nice write???

Just a single command and in an instant a tensor full of zeros and ones are created.

> Zeros are more commonly used then ones.

btw every spelling of zeroes is a spelling mistake 😁

Arange & Tensors Like

Arange

Now, there is some more functions that can be very helpfull in making a tensor.

If you need a tensor of continuous values you can use the `arange` function.

```
torch.arange(start=0,end=100)
```

```
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, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35,
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53,
        54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71,
        72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89,
        90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

And there you have a 1-dimensional tensor with from `0 to 99`. The `arange` function can take `2 arguments`

- **start**: This argument is for the starting value of the tensor.
- **end**: This is for the ending value of the tensor. The ending value is exclusive so the value given in the end argument will be excluded in the tensor, that's why the tensor we made have 0 to 99 even though we passed 100 as the end value.

To make a tensor with 0 to 100 values we have to make do the following.

```
torch.arange(start=0,end=101)
```

```
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,  24,  25,  26,
27,
         28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40,
41,
         42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,  53,  54,
55,
         56,  57,  58,  59,  60,  61,  62,  63,  64,  65,  66,  67,  68,
69,
         70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,  81,  82,
83,
         84,  85,  86,  87,  88,  89,  90,  91,  92,  93,  94,  95,  96,
97,
         98,  99, 100])
```

And we have a nice tensor.

Now there is another argument that the `arange` function can take. -**step**: Steps argument is used to specify the gap between the tensor values. It is set to 1 by default. YOu try to experiment by changing this parameter by padding a different value as an argument.

```
torch.arange(start=-,end=101,step=2)
```

Try it out and see what is the tensor you get.

And that will bring me to your next homework.

> 🗨️ Make two tensors, one will have all `odd` numbers from `0 to 100` , other one will have all `Even` numbers from `0 to 100`.

That brings us to the next part.

like

This is a helper function that can make a tesnor like other tensors.

For example you need a tensor of zeroes like the tensor we made earlier with arnge function.

> The tensor from 0 to 100.

The like functions can make a tensor with 100 zeros.

> Yes it is plural, there are other Like functions and all of them are written like this. (`FunctionName_Like`)

```

```

```
one_to_ten = torch.arange(start=1,end=11)

ten_zeros = torch.zeros_like(input=one_to_ten)
ten_zeros
```

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

There you have it it's fairly simple and anyone can master it as they practice more and more.

Tensor DataTypes

`Data Types` are a very important topic in any language or library.

🏅 There are `3` common error you will face using `Pytorch`:

- `Wrong Data Types`
- `Wrong Shape`
- `Wrong Device`

Let's see what type of `error` we can face and what is the `default` data type of a tensor.

I'll make a integer tensor.

```
int_tensor = torch.tensor([1,2,3,4])
```

Now we can see the data type of the tensor by doing the following.

```
int_tensor.dtype
```

```
torch.int64
```

Well well well, the tensor we created has a datatype of `torch.int64` meaning that the tensor have a limit of `2^64` bits. If you don't know how this works do your own research. You are just one `google search` away.

Homework Time.

🗨️ Research about `Torch DataTypes` in google also in the `torch official documentations`. Also Learn about the term `Precision in Computing`(This is directly related to TOrch DataTypes)

So, the limit is `int64` or `2^64` bits right??

I'm curious about what is the value of 2^64!?

```
2**64
```

```
18446744073709551616
```

This is a very huge number. So, in theory we can only store a value up to `18446744073709551616`. So, let's now make a new tensor with a larger value, for example: `18446744073709551800`.

```
new_tensor = torch.tensor([1,2,3,118446744073709551800])
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

Cell In[101], line 1
----> 1 new_tensor = torch.tensor([1,2,3,118446744073709551800])


RuntimeError: Overflow when unpacking long
```

An we have an `error`. This shows that a value of the tensor `Overflowing when unpacking long(int64)`. This is because we have a value that is `out of limit`.

Now what do we do, TO store this value larger then the limit we need to change the data type to a larger limit.

There are many data types we can use in pytorch. So, I'll very much suggest you go and have a look at the official documentation and you will know all about the datatypes.

Soooo, we have a big problem now. Pytorch does not have a `int` datatype larger then the `int64` so, what do we do?

! We can transform any `int` value into a `float` value and it will be stored. But some of the data might be lost.

Let's see what happend if we make the same tensor with a `float64` dtype.

```
new_tensor = torch.tensor([1,2,3,18446744073709551800],
dtype=torch.float64)
new_tensor
```

```
tensor([1.0000e+00, 2.0000e+00, 3.0000e+00, 1.8447e+19],
dtype=torch.float64)
```

AAAAANNNNNDD there you go we have a tensor that can store a value `larger then int64` by storing the data in `floating point` and we also didn't get a error message.

But there is another issue left. You can see that the large tensor value is stored like this, `1.887e+19` meaning `1.887*10^19`. So, there can be some data loss.

So, let's check if there is some data loss or not. We can compare the `int` value with the `float value` to see if they are the same.

```
18446744073709551800 == int(new_tensor[-1])
```

```
False
```

Oh!!! `It returned false`. So, the data we stored is not the same as we wanted to store. So, there is a possibily of data loss. We, how much data did we lose.

```
18446744073709551800 - int(new_tensor[-1])
```

```
184
```

The data we stored is `184` less then the data we wanted to store. This brings us to the first complication of `pytorch`.

> 🚫 There is no way to store data larger then `int64` and if you store the data using `float64` some data will be lost because `float64` also has the limit of `2^64` bits.

> 🗨️ Figure out how to convert a tensors data type without making a whole new tensor.

Matrix Multiplication

`Matrix Multiplication` is one of the most important topics in `deep learning` and machine learning. That brings me to the first MOTO of this article series.

> # THERE IS NO DEEP LEARNING WITHOUT DEEP MATHS KNOWLEDGE.

Yes! you need to know almost every mathematical theories you learned so far and much more and one of those mathematical theories is `Matrix`. SOOOOOO,

> 🗨 Research on `matrix` and how to do `matrix multiplications`.

a `2-dimensional` tensor is the representation of a `Matrix` in Deep leaning. Thats why at the starting of this article I made a 2-dimensional tensor and named it a `MATRIX`.

Now how can we muliply two matrixs.

Let's make two matrixs.

```
matrix_A = torch.randint(low=1,high=10,size=(10,))
matrix_B = torch.randint(low=1,high=10,size=(10,))
matrix_A,matrix_B
```

```
(tensor([6, 7, 6, 2, 3, 5, 7, 8, 1, 4]),
 tensor([4, 5, 6, 3, 3, 7, 9, 3, 2, 5]))
```

There you go! We have made two tensors with `10 random values from 1 to 10`. Now how do can we multiply these two tensors?

> ❗ There are two ways to multiply tensors.
>
> - `Value wise multiplication`.
> - `Big Dot or Matrix Multiplication`.

Value wise multiplication is multiplying two tensor indexes. Which can be done by simple `*` opration.

```
matrix_A*matrix_B
```

```
tensor([24, 35, 36,  6,  9, 35, 63, 24,  2, 20])
```

And we have a nice value wise multiplied tensor where every index is the multiplied value of the two tensor indecies.

Now what is `matrix Multiplication`?

As I said earlier this is a mathematical concept that is widely used in deep learning and I can write a dedicated article to just explain `what a matrix is`. But I will not go into mathematical details.

`Matrix Multiplication` can be done by a simple `torch command`.

```
torch.matmul(matrix_A,matrix_B)
```

```
tensor(254)
```

WHAT!!!!

`Matrix Multiplting` two tensors returned a single value???

How does that happen??

Calm down I will explain.

The output is currect But here is a question for you.

🗨️ What are the dimensions of these two tensors? are they really matrixs??

```
print(f"Dimensions of Matrix A: {matrix_A.ndim}")
print(f"Dimensions of Matrix B: {matrix_B.ndim}")
```

```
Dimensions of Matrix A: 1
Dimensions of Matrix B: 1
```

WHAAT!!! These two tensors was not even matrixs!!!!

> 1-dimesional tensors are called scalers.

Oh well, Im a shity article writter and obviously I made a huge error while making this article.

NOPE

> **!** Every 1-dimensional tensors can be intepreted as a (1xn) matrix. 1 is the number of rows and n is the number of elements or we can say number of columns.

As we made two 1-dimensional tensors, we can now make two legitimate matrixs two know what really happens.

> we can just reshape the 1-dimensional tensors from before.

```
matrix_A = matrix_A.reshape(shape=(1,10))
matrix_B = matrix_B.reshape(shape=(1,10))
```

```
torch.matmul(matrix_A,matrix_B)
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

Cell In[110], line 1
----> 1 torch.matmul(matrix_A,matrix_B)


RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x10 and 1x10)
```

WHAAAT!!!!!!!!

It doesn't work?! But It's the same matrix as the matrix we made before right?

Let's see the value and the number of dimesions of the `reshaped` tensors from before.

```
print(f"Matrix A: {matrix_A}")
print(f"Matrix B: {matrix_B}")
print(f'dimensions of A: {matrix_A.ndim}')
print(f'dimensions of B: {matrix_B.ndim}')
```

```
Matrix A: tensor([[6, 7, 6, 2, 3, 5, 7, 8, 1, 4]])
Matrix B: tensor([[4, 5, 6, 3, 3, 7, 9, 3, 2, 5]])
dimensions of A: 2
dimensions of B: 2
```

we have successfully reshaped the 1-dimensional tensors to matrixs. But why did we get the error?

> **!** There are cruicial rule we must follow:
>
> 1. The inner Dimensional shape values must be equal. Sooo, **(3,2)x(2,3)**✅ **(3,3)x(3,2)**✅ **(2,3)x(3,1)**✅ **(3,2)x(3,2)**❌ **(5,2)x(3,1)**❌

> **!** Inner dimensional shape value must equal means, to the `number columns of the first matrix` and the `number of rows of the second matrix` should be same.

So, what is the shape the matrixs we made earlier?

```
print(f"shape of A: {matrix_A.shape}")
print(f"shape of B: {matrix_B.shape}")
```

```
shape of A: torch.Size([1, 10])
shape of B: torch.Size([1, 10])
```

WHAAAT?! ops, this is an overreaction.

The shape of A is [1, 10] and shape of B [1, 10]. So, the number of columns our first matrix is `10` and the number of rows of our second matrix is `1`, which is clearly `not the same` so it is impossible to do a matrix multiplication between these two matrixs.

This `error` is the most common error in matrix multiplications.

> 🗨️ Figure out why the 1-dimensional tensors can be matrix multiplied without following the rules and what is a `Transposed Matrix` is?

This brings us to the another concept that cruisal in matrix multiplication -> `Transposing a Matrix`.

Pytorch has a built in function to do this. Let's transpose the matrix B.

```
transposed_matrix_B = matrix_B.T
transposed_matrix_B
```

```
tensor([[4],
        [5],
        [6],
        [3],
        [3],
        [7],
        [9],
        [3],
        [2],
        [5]])
```

> You can transpose a matrix using the `matrix_valiable_name.T` function.

Well that's weird. What happened there??

Every element in the row became their own rows.

Let's see the shape of this transposed matrix B

```
transposed_matrix_B.shape
```

```
torch.Size([10, 1])
```

Woah! the shape interchanged. The row value became the column value and column value became the row.

So, after transposing the second matrix are fullfilling the first rule.

- Shape of A is (1,10)
- Shape of transposed B is (10,1).
- The matrix we are multiplying has the shape `(1,10)x(10,1)`, The inner dimensions for both of these matrix are equal.

So, we should be able use the `torch.matmul()` function to multiply these two tensors.

Let's if we are right.

```
torch.mm(matrix_A,transposed_matrix_B)
```

```
tensor([[254]])
```

> torch.mm() can also be used for matrix multiplications.

AAAANNNNNNDDDD there you go. We have done our first matrix multiplication. We can do the same for the matrix A. This should give us the same result right?

```
matrix_A.T.shape, matrix_B.shape
```

```
(torch.Size([10, 1]), torch.Size([1, 10]))
```

> (10,1)x(1,10)

The inner dimensional values are equal. Let's multiply.

```
torch.mm(matrix_A.T, matrix_B)
```

```
tensor([[24, 30, 36, 18, 18, 42, 54, 18, 12, 30],
        [28, 35, 42, 21, 21, 49, 63, 21, 14, 35],
        [24, 30, 36, 18, 18, 42, 54, 18, 12, 30],
        [ 8, 10, 12,  6,  6, 14, 18,  6,  4, 10],
        [12, 15, 18,  9,  9, 21, 27,  9,  6, 15],
        [20, 25, 30, 15, 15, 35, 45, 15, 10, 25],
        [28, 35, 42, 21, 21, 49, 63, 21, 14, 35],
        [32, 40, 48, 24, 24, 56, 72, 24, 16, 40],
        [ 4,  5,  6,  3,  3,  7,  9,  3,  2,  5],
        [16, 20, 24, 12, 12, 28, 36, 12,  8, 20]])
```

WHAAAAAAAAAAT!!!!

What is this???????

Why is the result of these two matrix multiplications are different and why is the shape is of these multiplications are all so confusing?

- **RULE 2**: The outcome of the matrix multiplications will have the `shape of the outer dimensional values`. for example:

```
Input matrix shapes: (3,2), (2,3)
output shape: (3,3)

because the outer shapes are 3 and 3
```

> outer shapes refers to the number of rows of the first matrix and the number of columns of the second matrix.

In our case, after transposing B the shapes were (1,10) and (10,1). Rule 1 is followed.

The outer dimensions are `1 and 1`. So the output shape would be (1,1). Let's see the proof.

```
torch.matmul(matrix_A,matrix_B.T).shape
```

```
torch.Size([1, 1])
```

And we are right.

For the second matmul, we have the shape of (10,1) and (1,10). So, the output shape should be `(10,10)`.

```
torch.mm(matrix_A.T,matrix_B).shape
```

```
torch.Size([10, 10])
```

AAAAANNNNNDDDD we are done.

That's it for the matmul section. This is all I know of the Matrix Multiplication using torch.

But I have to say, the article is still not done yet. There are some other small concepts left and I'll make small example for those try those out yourself and see what they really do.

For all those I'll make a dummy 4-dimensional tensor.

```

```

```
dummy = torch.randint(low=1, high=1000, size=(2,1,3,4))
dummy
```

```
tensor([[[[837, 325,  28, 115],
          [956, 692, 599, 659],
          [453, 211, 722, 125]]],


        [[[338, 746, 141, 835],
          [640, 870, 117, 942],
          [740, 188, 465, 632]]]])
```

Max

Max returns the maximum value in a tensor.

```
torch.max(dummy),dummy.max()
```

```
(tensor(956), tensor(956))
```

Min

Min returns the minimum value of a tensor.

```
torch.min(dummy), dummy.min()
```

```
(tensor(28), tensor(28))
```

Mean

The average of a tensor.

```
torch.mean(dummy), dummy.mean()
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

Cell In[123], line 1
----> 1 torch.mean(dummy), dummy.mean()


RuntimeError: mean(): could not infer output dtype. Input dtype must be
either a floating point or complex dtype. Got: Long
```

The error massege says that it's a data type error. So, we can change the datatype to `floating point`.

```
torch.mean(dummy.type(torch.float32)), dummy.type(torch.float32).mean()
```

```
(tensor(515.6667), tensor(515.6667))
```

ArgMax

Returns the position of the largest value.

```
torch.argmax(dummy), dummy.argmax()
```

```
(tensor(4), tensor(4))
```

Argmin

Returns the position of the smallest value in the tensor.

```
torch.argmin(dummy), dummy.argmin()
```

```
(tensor(2), tensor(2))
```

Squeeze

Removes the dimension with the shape of 1.

```
print(f'shape of the dummy tensor: {dummy.shape}')
print(f'shape of squeezed dummy tensor:
{dummy.squeeze().shape,torch.squeeze(dummy).shape}')
```

```
shape of the dummy tensor: torch.Size([2, 1, 3, 4])
shape of squeezed dummy tensor: (torch.Size([2, 3, 4]), torch.Size([2, 3,
4]))
```

The 4 dimensional tensor became 3-dimensional because of squeeze.

Unsqueeze

Add a dimension in the position of your choice.

```
print(f'shape of the dummy tensor: {dummy.shape}')
print(f'shape of unsqueezed dummy tensor: {dummy.unsqueeze(3).shape,
torch.unsqueeze(dummy, dim=(3)).shape}')
```

```
shape of the dummy tensor: torch.Size([2, 1, 3, 4])
shape of unsqueezed dummy tensor: (torch.Size([2, 1, 3, 1, 4]),
torch.Size([2, 1, 3, 1, 4]))
```

Permute

Rearranging the dimension.

```
dummy.permute(dims=(2,1,0,3)).shape
```

```
torch.Size([3, 1, 2, 4])
```

> 🗨️ Read the doc in the official pytorch website to have more in depth knowledge of these useful functions.

OKAY!!!

You have done it and reached the end of the this article and I hope you have learned a lot of new things on pytorch.

The next article we will make our first Deep leaning model and learn a lot of deep knowledge about Deep learning.

Stay tuned and HAPPY CODING.