Pytorch_Introduction

April 27, 2020

1 Introduction to Pytorch

This notebook will provide a brief overview of PyTorch and how it is similar to Numpy. The goal of this notebook is to understand the basic data structures required to build Deep Learning models and train them.

1.0.1 Why do we need PyTorch when we already have Numpy?

Deep Learning involves performing similar operations like convolutions and multiplications repetitively. Thus there is a need to run the code on GPUs which can parallelize these operations over multiple cores - these devices are perfectly suitable for doing massive matrix operations and are much faster than CPUs.

On the diagram below you can find iteration times for several well-known architectures of neural networks running on different hardware. Please note, that the vertical axis has a logarithmic scale. Data for this chart was from taken here.

While NumPy with its various backends suits perfectly for doing calculus on CPU, it lacks the GPU computations support. And this is the first reason why we need Pytorch.

The other reason is that Numpy is a genral purpose library. PyTorch (or any other modern deep learning library) has optimized code for many deep learning specific operations (e.g. Gradient calculations) which are not present in Numpy.

So, let's take a look at what are the data structures of Pytorch and how it provides us its cool features.

2 Tensor - Pytorch's core data structure

As we've already seen, in python we can create lists, lists of lists, lists of lists of lists and so on. In NumPy there is a `numpy.ndarray` which represents `n`-dimensional array, e.g. 3-dimensional if we want to find an analog for a list of lists of lists. In math, there is a special name for the generalization of vectors and matrices to a higher dimensional space - a tensor.

Tensor is an entity with a defined number of dimensions called an order (rank).

Scalars can be considered as a rank-0-tensor. Let's denote scalar value as $x \in \mathbb{R}$, where $\mathbb{R}$ is a set of real numbers.

Vectors can be introduced as a rank-1-tensor. Vectors belong to linear space (vector space), which is, in simple terms, a set of possible vectors of a specific length. A vector consisting of real-valued scalars ($x \in \mathbb{R}$) can be defined as ($y \in \mathbb{R}^n$), where y - vector value and $\mathbb{R}^n$ - n-dimensional real-number vector space. y_i - i_{th} vector element (scalar):

$$y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Matrices can be considered as a rank-2-tensor. A matrix of size $m \times n$, where $m, n \in \mathbb{N}$ (rows and columns number accordingly) consisting of real-valued scalars can be denoted as $A \in \mathbb{R}^{m \times n}$, where $\mathbb{R}^{m \times n}$ is a real-valued $m \times n$-dimensional vector space:

$$A = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

In the picture below you can find different tensor dimensions, where the fourth and the fifth cases are a vector of cubes and a matrix of cubes accordingly:

```
<tr><th><center>Rank-1-tensor</center></th><th><center>Rank-2-tensor</center></th><th><center>
    <th><center><img align="middle" src="https://www.learnopencv.com/wp-content/uploads/2020/0
    <th><center><img align="middle" src="https://www.learnopencv.com/wp-content/uploads/2020/0
    <th><center><img align="middle" src="https://www.learnopencv.com/wp-content/uploads/2020/0
    <th><center><img align="middle" src="https://www.learnopencv.com/wp-content/uploads/2020/0
    <th><center><img align="middle" src="https://www.learnopencv.com/wp-content/uploads/2020/0
```

2.1 Tensor basics

Let's import the torch module first

```
In [1]: import numpy as np
        import torch
```

2.1.1 Tensor Creation

Let's view examples of matrices and tensors generation.

2-dimensional (rank-2) tensor of zeros:

```
In [2]: torch.zeros(3, 4)

Out[2]: tensor([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])
```

Random rank-4 tensor:

```
In [3]: torch.rand(2, 2, 2, 2)

Out[3]: tensor([[[[0.6996, 0.2302],
                  [0.3362, 0.1222]],
                 [[0.7585, 0.4373],
                  [0.5245, 0.3768]]],


                [[[0.9012, 0.5454],
                  [0.9551, 0.7257]],

                 [[0.0114, 0.8099],
                  [0.8467, 0.2301]]]])
```

For sure, there are many more ways to create tensor using some restrictions on values it should contain - for the full reference, please follow the official docs.

2.1.2 Python / NumPy / Pytorch interoperability

You can create tensors from python lists as well as numpy arrays. You can also convert torch tensors to numpy arrays. So, the interoperability between torch and numpy is pretty good.

```
In [4]: # Simple Python list
        python_list = [1, 2]

        # Create a numpy array from python list
        numpy_array = np.array(python_list)

        # Create a torch Tensor from python list
        tensor_from_list = torch.tensor(python_list)

        # Create a torch Tensor from Numpy array
        tensor_from_array = torch.tensor(numpy_array)

        # Another way to create a torch Tensor from Numpy array (Share same storage)
        tensor_from_array_v2 = torch.from_numpy(numpy_array)

        # Convert torch tensor to numpy array
        array_from_tensor = tensor_from_array.numpy()

        print('List:   ', python_list)
        print('Array:  ', numpy_array)
        print('Tensor: ', tensor_from_list)
        print('Tensor: ', tensor_from_array)
        print('Tensor: ', tensor_from_array_v2)
        print('Array:  ', array_from_tensor)
```

```
List:    [1, 2]
Array:   [1 2]
Tensor:  tensor([1, 2])
Tensor:  tensor([1, 2])
Tensor:  tensor([1, 2])
```

```
Array:   [1 2]
```

Differnce between `torch.Tensor` and `torch.from_numpy` Pytorch aims to be an effective library for computations. What does it mean? It means that pytorch avoids memory copying if it can:

```
In [5]: numpy_array[0] = 10

        print('Array:  ', numpy_array)
        print('Tensor: ', tensor_from_array)
        print('Tensor: ', tensor_from_array_v2)
```

```
Array:   [10  2]
Tensor:  tensor([1, 2])
Tensor:  tensor([10,  2])
```

So, we have two different ways to create tensor from its NumPy counterpart - one copies memory and another one shares the same underlying storage. It also works in the opposite way:

```
In [6]: array_from_tensor = tensor_from_array.numpy()
        print('Tensor: ', tensor_from_array)
        print('Array: ', array_from_tensor)

        tensor_from_array[0] = 11
        print('Tensor: ', tensor_from_array)
        print('Array: ', array_from_tensor)
```

```
Tensor:  tensor([1, 2])
Array:  [1 2]
Tensor:  tensor([11,  2])
Array:  [11  2]
```

2.1.3 Data types

The basic data type for all Deep Learning-related operations is float, but sometimes you may need something else. Pytorch supports different numbers types for its tensors the same way as NumPy does it - by specifying the data type on tensor creation or via casting. The full list of supported data types can be found here.

```
In [7]: tensor = torch.zeros(2, 2)
        print('Tensor with default type: ', tensor)
        tensor = torch.zeros(2, 2, dtype=torch.float16)
        print('Tensor with 16-bit float: ', tensor)
        tensor = torch.zeros(2, 2, dtype=torch.int16)
        print('Tensor with integers: ', tensor)
        tensor = torch.zeros(2, 2, dtype=torch.bool)
        print('Tensor with boolean data: ', tensor)
```

```
Tensor with default type:  tensor([[0., 0.],
        [0., 0.]])
Tensor with 16-bit float:  tensor([[0., 0.],
        [0., 0.]], dtype=torch.float16)
Tensor with integers:  tensor([[0, 0],
        [0, 0]], dtype=torch.int16)
Tensor with boolean data:  tensor([[False, False],
        [False, False]])
```

2.1.4 Indexing

Tensor provides access to its elements via the same [] operation as a regular python list or NumPy array. However, as you may recall from NumPy usage, the full power of math libraries is accessible only via vectorized operations, i.e. operations without explicit looping over all vector elements in python and using implicit optimized loops in C/C++/CUDA/Fortran/etc. available via special functions calls. Pytorch employs the same paradigm and provides a wide range of vectorized operations. Let's take a look at some examples.

Joining a list of tensors together with `torch.cat`

```
In [8]: a = torch.zeros(3, 2)
        b = torch.ones(3, 2)
        print(torch.cat((a, b), dim=0))
        print(torch.cat((a, b), dim=1))
```

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

Indexing with another tensor/array:

```
In [9]: a = torch.arange(start=0, end=10)
        indices = np.arange(0, 10) > 5
        print(a)
        print(indices)
        print(a[indices])

        indices = torch.arange(start=0, end=10) % 5
        print(indices)
        print(a[indices])
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
[False False False False False False  True  True  True  True]
tensor([6, 7, 8, 9])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
tensor([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

What should we do if we have, say, rank-2 tensor and want to select only some rows?

```
In [10]: tensor = torch.rand((5, 3))
         rows = torch.tensor([0, 2, 4])
         tensor[rows]
```

```
Out[10]: tensor([[0.6175, 0.3933, 0.5897],
                 [0.4305, 0.1123, 0.2643],
                 [0.6490, 0.9308, 0.3763]])
```

2.1.5 Tensor Shapes

Reshaping a tensor is a frequently used operation. We can change the shape of a tensor without the memory copying overhead. There are two methods for that: `reshape` and `view`.

The difference is the following: - view tries to return the tensor, and it shares the same memory with the original tensor. In case, if it cannot reuse the same memory due to some reasons, it just fails. - reshape always returns the tensor with the desired shape and tries to reuse the memory. If it cannot, it creates a copy.

Let's see with the help of an example

```
In [11]: tensor = torch.rand(2, 3, 4)
         print('Pointer to data: ', tensor.data_ptr())
         print('Shape: ', tensor.shape)

         reshaped = tensor.reshape(24)

         view = tensor.view(3, 2, 4)
         print('Reshaped tensor - pointer to data', reshaped.data_ptr())
         print('Reshaped tensor shape ', reshaped.shape)

         print('Viewed tensor - pointer to data', view.data_ptr())
         print('Viewed tensor shape ', view.shape)

         assert tensor.data_ptr() == view.data_ptr()

         assert np.all(np.equal(tensor.numpy().flat, reshaped.numpy().flat))

         print('Original stride: ', tensor.stride())
         print('Reshaped stride: ', reshaped.stride())
         print('Viewed stride: ', view.stride())
```

```
Pointer to data:  48221760
Shape:  torch.Size([2, 3, 4])
Reshaped tensor - pointer to data 48221760
Reshaped tensor shape  torch.Size([24])
Viewed tensor - pointer to data 48221760
Viewed tensor shape  torch.Size([3, 2, 4])
Original stride:  (12, 4, 1)
Reshaped stride:  (1,)
Viewed stride:  (8, 4, 1)
```

The basic rule about reshaping the tensor is definitely that you cannot change the total number of elements in it, so the product of all tensor's dimensions should always be the same. It gives us the ability to avoid specifying one dimension when reshaping the tensor - Pytorch can calculate it for us:

```
In [12]: print(tensor.reshape(3, 2, 4).shape)
         print(tensor.reshape(3, 2, -1).shape)
         print(tensor.reshape(3, -1, 4).shape)
```

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

Alternative ways to view tensors - `expand` or `expand_as`.

- `expand` - requires the desired shape as an input
- `expand_as` - uses the shape of another tensor.

These operaitions "repeat" tensor's values along the specified axes without actual copying the data.

As the documentation says, expand > returns a new view of the self tensor with singleton dimensions expanded to a larger size. Tensor can be also expanded to a larger number of dimensions, and the new ones will be appended at the front. For the new dimensions, the size cannot be set to -1.

Use case:

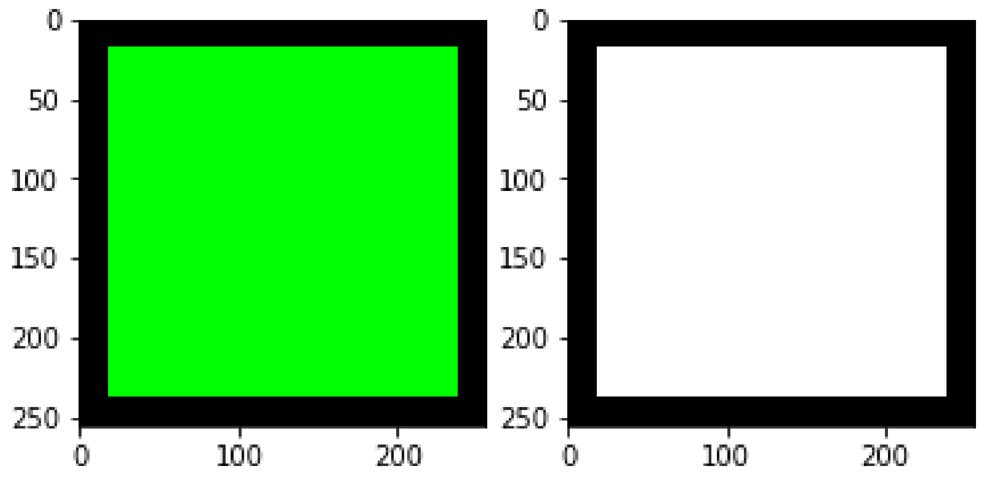
- index multi-channel tensor with single-channel mask - imagine a color image with 3 channels (R, G and B) and binary mask for the area of interest on that image. We cannot index the image with this kind of mask directly since the dimensions are different, but we can use `expand_as` operation to create a view of the mask that has the same dimensions as the image we want to apply it to, but has not copied the data.

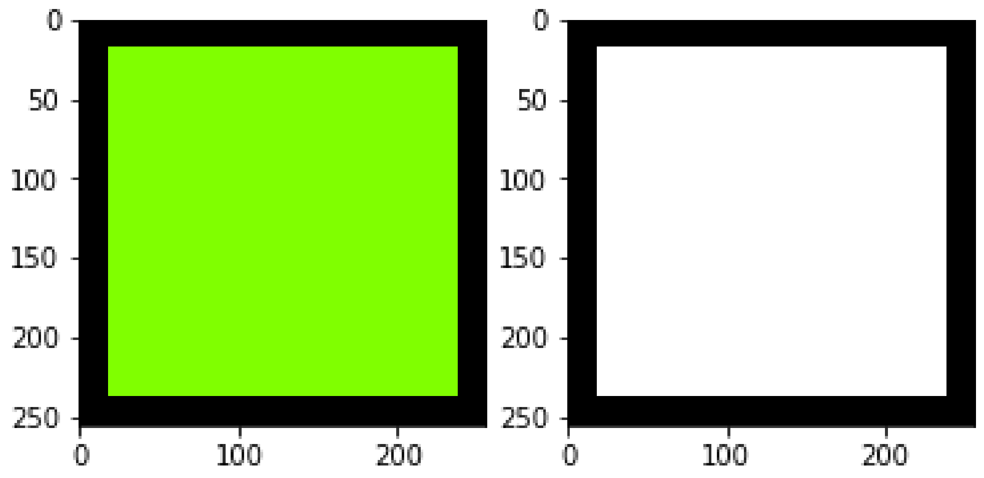
```
In [13]: %matplotlib inline
         from matplotlib import pyplot as plt

         # Create a black image
         image = torch.zeros(size=(3, 256, 256), dtype=torch.int)
```

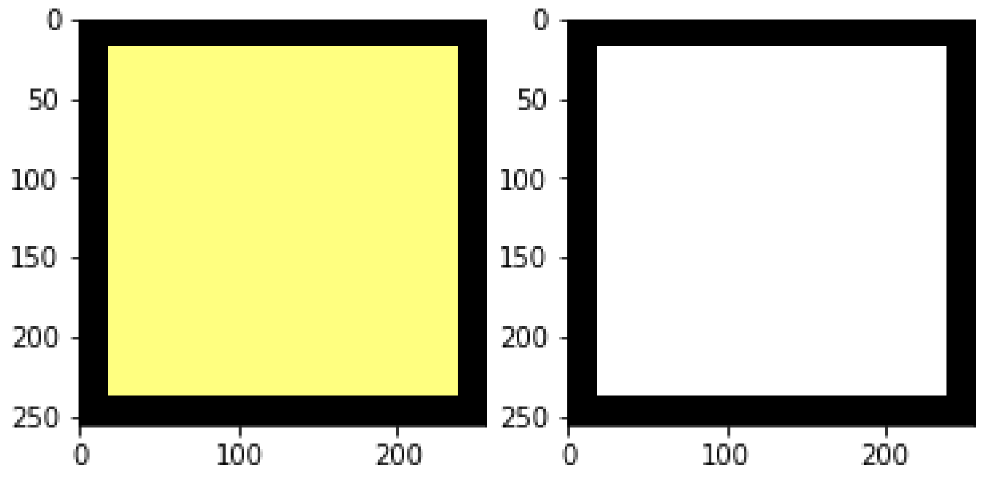
```
        # Leave the borders and make the rest of the image Green
        image[1, 18:256 - 18, 18:256 - 18] = 255

        # Create a mask of the same size
        mask = torch.zeros(size=(256, 256), dtype=torch.bool)

        # Assuming the green region in the original image is the Region of interest, change t
        mask[18:256 - 18, 18:256 - 18] = 1

        # Create a view of the mask with the same dimensions as the original image
        mask_expanded = mask.expand_as(image)
        print(mask_expanded.shape)

        mask_np = mask_expanded.numpy().transpose(1, 2, 0) * 255
        image_np = image.numpy().transpose(1, 2, 0)

        fig, ax = plt.subplots(1, 2)
        ax[0].imshow(image_np)
        ax[1].imshow(mask_np)
        plt.show()

        image[0, mask] += 128
        fig, ax = plt.subplots(1, 2)
        ax[0].imshow(image_np)
        ax[1].imshow(mask_np)
        plt.show()

        image[mask_expanded] += 128
        image.clamp_(0, 255)
        fig, ax = plt.subplots(1, 2)
        ax[0].imshow(image_np)
        ax[1].imshow(mask_np)
        plt.show()
```

```
Matplotlib is building the font cache using fc-list. This may take a moment.
```

```
torch.Size([3, 256, 256])
```







In the example above, one can also find a couple of useful tricks: - `clamp` method and function is a Pytorch's analogue of NumPy's `clip` function - many operations on tensors have in-place form, that does not return modified data, but change values in the tensor. The in-place version of the operation has trailing underscore according to Pytorch's naming convention - in the example above it is `clamp_` - tensors have the same indexing as Numpy's arrays - one can use `:`-separated range, negative indexes and so on.

3 Images and their representations

Now, let's discuss images, their representations and how different Python libraries work with them.

Probably, the most well-known library for image loading and simple processing is Pillow. Another common alternative, that supports lots of image (and not only image) formats is imageio.

However, many people in deep learning area stick with OpenCV for image loading and processing with some usage of another libraries when it is justified by performance/functionality. This is because OpenCV is in general much faster than the other libraries. Here you can find a couple of benchmarks: - https://www.kaggle.com/zfturbo/benchmark-2019-speed-of-image-reading - https://github.com/albumentations-team/albumentations#benchmarking-results

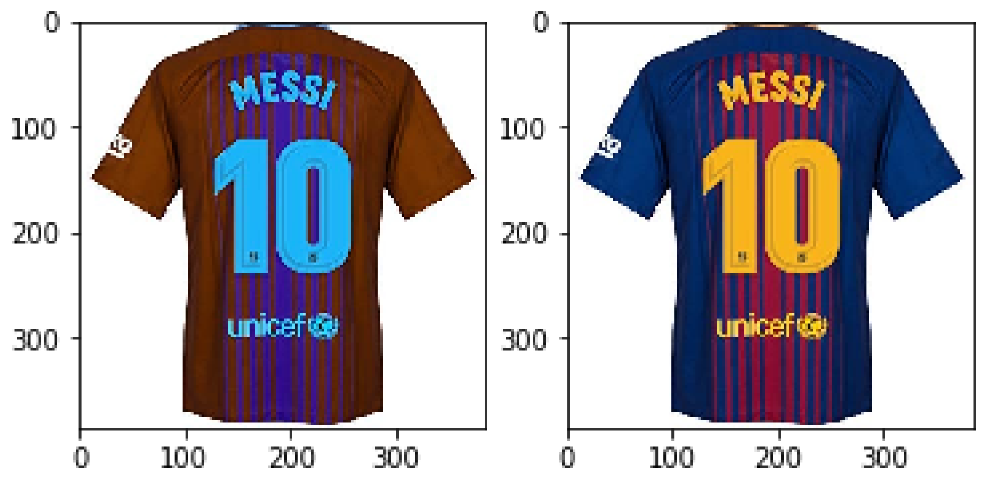
To sum up the benchmarks above, there are two most common image formats: PNG and JPEG. If your data is in PNG format - use OpenCV to read it. If it is in JPEG - use libturbojpeg. For image processing, use OpenCV if possible.

However, you can find some pitfalls in these image reading/showing/processing operations. Let's discuss them.

As you will read the code from other some, you may find out that some of them still use Pillow/something else to read data. You should know, that color image representations in OpenCV and other libraries are different - OpenCV uses "BGR" channel order, while others use "RGB" one. Good news is that it's easy to convert one representation to another and vice versa.

To cope with "BGR" <-> "RGB" conversion let's recall that image is actually a tensor with the shape `HxWxC`, where `H` is the height of the image, `W` is its width and `C` is the number of channels, in our case - 3. To convert image from "BGR" representation to "RGB" the only thing we should do is to change channel order!

```
In [14]: %matplotlib inline
         from matplotlib import pyplot as plt
         import cv2

         bgr_image = cv2.imread('../resource/lib/publicdata/images/jersey.jpg')
         rgb_image = bgr_image[..., ::-1]
         fig, ax = plt.subplots(1, 2)
         ax[0].imshow(bgr_image)
         ax[1].imshow(rgb_image)
         plt.show()
```



Please note that Matplotlib also uses "RGB" channel order, to the right cat image is correct.

4 Autograd

Pytorch supports automatic differentiation. The module which implements this is called **Autograd**. It calculates the gradients and keeps track in forward and backward passes. For primitive tensors, you need to enable or disable it using the `requires_grad` flag. But, for advanced tensors, it is enabled by default.

```
In [15]: a = torch.rand((3, 5), requires_grad=True)
         print(a)
         result = a * 5
         print(result)

         # grad can be implicitly created only for scalar outputs
         # so let's calculate the sum here so that the output becomes a scalar and we can appl
         mean_result = result.sum()
         # Calculate Gradient
         mean_result.backward()
         # Print gradient of a
         print(a.grad)
```

```
tensor([[0.6247, 0.2995, 0.3239, 0.9206, 0.7075],
        [0.8187, 0.4152, 0.9893, 0.6430, 0.2024],
        [0.3504, 0.4021, 0.5422, 0.7064, 0.2054]], requires_grad=True)
tensor([[3.1234, 1.4976, 1.6196, 4.6030, 3.5376],
        [4.0934, 2.0762, 4.9466, 3.2149, 1.0120],
        [1.7520, 2.0103, 2.7110, 3.5322, 1.0271]], grad_fn=<MulBackward0>)
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]])
```

As we see, Pytorch automagically calculated the gradient value for us. It looks to be the correct value - we multiplied an input by 5, so the gradient of this operation equals to 5.

4.0.1 Disabling Autograd for tensors

We don't need to compute gradients for all the variables that are involved in the pipeline. The Pytorch API provides 2 ways to disable autograd.

1. `detach` - returns a copy of the tensor with autograd disabled. This copy is built on the same memory as the original tensor, so in-place size / stride / storage changes (such as resize_ / resize_as_ / set_ / transpose_) modifications are **not** allowed.
2. `torch.no_grad()` - It is a context manager that allows you to guard a series of operations from autograd without creating new tensors.

```
In [16]: a = torch.rand((3, 5), requires_grad=True)
         detached_a = a.detach()
         detached_result = detached_a * 5
         result = a * 10
         # we cannot do backward pass that is required for autograd using multideminsional out
         # so let's calculate the sum here
         mean_result = result.sum()
         mean_result.backward()
         a.grad

Out[16]: tensor([[10., 10., 10., 10., 10.],
                 [10., 10., 10., 10., 10.],
                 [10., 10., 10., 10., 10.]])
```

```
In [17]: a = torch.rand((3, 5), requires_grad=True)
         with torch.no_grad():
             detached_result = a * 5
         result = a * 10
         # we cannot do backward pass that is required for autograd using multideminsional out
         # so let's calculate the sum here
         mean_result = result.sum()
         mean_result.backward()
         a.grad

Out[17]: tensor([[10., 10., 10., 10., 10.],
                 [10., 10., 10., 10., 10.],
                 [10., 10., 10., 10., 10.]])
```