



General Sir John Kotelawala Defense University

B.Sc. (Hons) Data Science and Business Analytics

INTAKE 38 SEMESTER VII

CS4192- Deep Learning

Student Name	Reg no
D N R Peiris	D/DBA/21/0026
Lecturer in-charge Mr. Gihan Liyanage	

Table of Contents

1 What is TORCH.TENSOR?	3
2 Tensor Data Types	3
3 Tensor Functions	4
3.1 Addition	4
3.2 Subtraction	5
3.3 Elementwise Multiplication	5
3.4 Matrix Multiplication	5
3.5 Transpose	6
3.6 Elementwise Division	6
3.7 TORCH.ARANGE	6
3.8 TORCH.SPSPACE_COO_TENSOR	7
3.9 TORCH.NUMEL	9
3.10 Elementwise Exponential	9
3.11 Concatenation	9
3.12 Reshape the Tensor	10
3.13 Elementwise less than and Greater than	10
3.14 Calculate the determinate of a Matrix.	11
3.15 TORCH.FULL	11
3.16 TORCH.MASKED_SELECTED	12
3.17 Batch Matrix Multiplication	12
3.18 TORCH.VSPLIT	14
3.19 Repeat Tensor Along with the Dimension	14
3.20 Cumulative Product along the Dimension	15
3.21 TORCH.PERMUTE	16
3.22 TORCH.WHERE	17
3.23 TORCH.NN.FUNCTIONAL.PAD	18
3.24 TORCH.ARGMAX	18
3.25 TORCH.AUTOGRAD.GRAD	19
References	20
Appendix	20

1 What is TORCH.TENSOR?

A torch.Tensor is a multi-dimensional matrix which containing elements of single data. According to the PyTorch documentation, tensors are defined as a data structure which are really similar to the arrays and matrices. In PyTorch tensors used to encode the inputs and outputs of the model and model's parameters. Tensors are similar to NumPy's and ndarrays except they can run on GPUs and hardware accelerators.

2 Tensor Data Types

Torch defines there are 10 types with CPU and GPU variants.

The following table is taken from the PyTorch Documentation.

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	<code>torch.float32</code> or <code>torch.float</code>	<code>torch.FloatTensor</code>	<code>torch.cuda.FloatTensor</code>
64-bit floating point	<code>torch.float64</code> or <code>torch.double</code>	<code>torch.DoubleTensor</code>	<code>torch.cuda.DoubleTensor</code>
16-bit floating point	<code>torch.float16</code> or <code>torch.half</code>	<code>torch.HalfTensor</code>	<code>torch.cuda.HalfTensor</code>
16-bit floating point	<code>torch.bfloat16</code>	<code>torch.BFloat16Tensor</code>	<code>torch.cuda.BFloat16Tensor</code>
32-bit complex	<code>torch.complex32</code> or <code>torch.chalf</code>		
64-bit complex	<code>torch.complex64</code> or <code>torch.cfloat</code>		
128-bit complex	<code>torch.complex128</code> or <code>torch.cdouble</code>		
8-bit integer (unsigned)	<code>torch.uint8</code>	<code>torch.ByteTensor</code>	<code>torch.cuda.ByteTensor</code>
8-bit integer (signed)	<code>torch.int8</code>	<code>torch.CharTensor</code>	<code>torch.cuda.CharTensor</code>
16-bit integer (signed)	<code>torch.int16</code> or <code>torch.short</code>	<code>torch.ShortTensor</code>	<code>torch.cuda.ShortTensor</code>
32-bit integer (signed)	<code>torch.int32</code> or <code>torch.int</code>	<code>torch.IntTensor</code>	<code>torch.cuda.IntTensor</code>
64-bit integer (signed)	<code>torch.int64</code> or <code>torch.long</code>	<code>torch.LongTensor</code>	<code>torch.cuda.LongTensor</code>
Boolean	<code>torch.bool</code>	<code>torch.BoolTensor</code>	<code>torch.cuda.BoolTensor</code>
quantized 8-bit integer (unsigned)	<code>torch.quint8</code>	<code>torch.ByteTensor</code>	/
quantized 8-bit integer (signed)	<code>torch.qint8</code>	<code>torch.CharTensor</code>	/
quantized 32-bit integer (signed)	<code>torch.qint32</code>	<code>torch.IntTensor</code>	/
quantized 4-bit integer (unsigned)	<code>torch.quint4x2</code>	<code>torch.ByteTensor</code>	/

3 Tensor Functions

To perform several tensor functions, first two tensors called tensor_a and tensor_b was created.

```
[ ] 1 import torch

1 #Creating Tensors
2 tensor_a = torch.tensor([[1,2],[3,4]])
3 tensor_b = torch.tensor([[5,6],[7,8]])

[ ] 1 print(f'Tensor A: ',tensor_a,'\nTensor B:',tensor_b)

Tensor A: tensor([[1, 2],
                  [3, 4]])
Tensor B: tensor([[5, 6],
                  [7, 8]])
```

3.1 Addition

Using torch.add() function, addition operation was performed on above created tensors.

```
[ ] 1 # Addition
2 addition_results = torch.add(tensor_a,tensor_b)
3 print(f'Addition of the Tensor A and Tensor B is: ',addition_results)

Addition of the Tensor A and Tensor B is: tensor([[ 6,  8],
          [10, 12]])
```

In here first element of tensor_a matrix and first element of tensor_b (number 1 and 5) are added and given as 6. Subsequently, four elements in tensor_a and four elements in tensor_b are added and given as [6, 8] and [10, 12] as addition_result tensor.

```
1 #Creating Tensors
2 tensor_a = torch.tensor([[1,2],[3,4]])
3 tensor_b = torch.tensor([[5,6],[7,8]])
```

3.2 Subtraction

Using torch.sub() function subtraction was performed on above tensors.

```
1 # Subtraction
2 subtraction_results = torch.sub(tensor_a, tensor_b)
3 print(f'Subtraction of the Tensor A and Tensor B is: ', subtraction_results)
```

Subtraction of the Tensor A and Tensor B is: tensor([[-4, -4],
[-4, -4]])

Similarly to addition, subtraction going through first element of tensor_a and first element of tensor_b, then subsequently goes through other elements of the tensors.

3.3 Elementwise Multiplication

Using torch.mul() function, multiplies the corresponding elements of tensor_a and tensor_b.

```
1 # Element wise Multiplication
2 element_wise_multi = torch.mul(tensor_a, tensor_b)
3 print(f'Elementwise Multiplication of the Tensor A and Tensor B is: ', element_wise_multi)
```

Elementwise Multiplication of the Tensor A and Tensor B is: tensor([[5, 12],
[21, 32]])

3.4 Matrix Multiplication

Consider tensor_a and tensor_b as matrix format mathematically.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

When perform torch.mm() function on tensor_a and tensor_b the mat_mul_result tensor perform matrix multiplication on tensor_a and tensor_b as follows. Let's consider the results stored in variable C.

$$C = \begin{bmatrix} (1.5 + 2.7) & (1.6 + 2.8) \\ (3.5 + 4.7) & (3.6 + 4.8) \end{bmatrix}$$

3.5 Transpose

`Torch.transpose(tensor_a,0,1)` change the dimension of the tensor_a

```
1 #Transpose
2 transpose_result = torch.transpose(tensor_a, 0, 1)
3 print(f'Transpose of the Tensor is: ',transpose_result)

Transpose of the Tensor is:  tensor([[1, 3],
                                     [2, 4]])
```

For simplicity transpose function behave like bellow,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

3.6 Elementwise Division

Divide corresponding elements of two tensors.

```
1 # Elementwise Division
2 elementwise_divide_result = torch.div(tensor_a, tensor_b.float())
3 print(f'Elementwise division of tensor A and Tensor B is: ',elementwise_divide_result)

Elementwise division of tensor A and Tensor B is:  tensor([[0.2000, 0.3333],
                                     [0.4286, 0.5000]])
```

3.7 TORCH.ARANGE

Returns 1-D tensor of size $\left\lceil \frac{end-start}{step} \right\rceil$ with values from the interval $[start, end)$ taken with common difference step beginning from the start

As an example, let's take `torch.arange(5)`,

```
1 # TORCH.ARANGE
2 torch.arange(5)

tensor([0, 1, 2, 3, 4])
```

When we consider the size of the 1-D tensor,

Start = 0 , end =5 and step=1 (gap between adjacent points)

Therefore, size of the tensor is, $size = \left\lceil \frac{5-0}{1} \right\rceil = 5$

As few examples bellow,

```
3 torch.arange(1,10)
tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
4 torch.arange(1,10,0.5)
tensor([1.0000, 1.5000, 2.0000, 2.5000, 3.0000, 3.5000, 4.0000, 4.5000, 5.0000,
        5.5000, 6.0000, 6.5000, 7.0000, 7.5000, 8.0000, 8.5000, 9.0000, 9.5000])
```

3.8 TORCH.SPSPACE_COO_TENSOR

This function is used to create a sparse tensor in the Coordinate Format (COO). These sparse tensors are useful when dealing with a large matrix which containing many zero elements, they cause to increase the significant amount of memory usage and computational time.

Following example which contained in PyTorch documentation explain the behavior of the sprace function.

```

1  i = torch.tensor([[0, 1, 1],
2  | | | | | | | | [2, 0, 2]])
3  v = torch.tensor([3, 4, 5], dtype=torch.float32)
4
5  # Create a sparse COO tensor with specified size [2, 4]
6  sparse_coo_tensor_1 = torch.sparse_coo_tensor(i, v, [2, 4])
7
8  print("Sparse COO Tensor 1:")
9  print(sparse_coo_tensor_1)
10
11 # Create a sparse COO tensor without specifying size
12 sparse_coo_tensor_2 = torch.sparse_coo_tensor(i, v)
13 print("\nSparse COO Tensor 2:")
14 print(sparse_coo_tensor_2)
15

```

```

Sparse COO Tensor 1:
tensor(indices=tensor([[0, 1, 1],
                       [2, 0, 2]]),
       values=tensor([3., 4., 5.]),
       size=(2, 4), nnz=3, layout=torch.sparse_coo)

```

```

Sparse COO Tensor 2:
tensor(indices=tensor([[0, 1, 1],
                       [2, 0, 2]]),
       values=tensor([3., 4., 5.]),
       size=(2, 3), nnz=3, layout=torch.sparse_coo)

```

1. “i” is the tensor which containing non-zero elements. In this example “i” has 2 rows and 3 columns.
2. “v” tensor containing the values corresponding to the non-zero elements at the specified indices. The values are of type float32.
3. “**torch.sparse_coo_tensor(i, v, [2, 4])**” this part creates a sparse COO tensor using the provided indices ‘i’ values. And the specified size [2,4], which means given sparse tensor have 2 rows and 4 columns. **The non-zero values are placed at specified indices with corresponding value.**

For the second sparse tensor,

1. The ‘i’ tensor contains the indices of non-zero elements. In the given example, the maximum index values along each dimension are 2 and 2.

2. When creating the sparse COO tensor using `torch.sparse_coo_tensor(i, v)`, PyTorch infers the size of the sparse tensor based on the maximum index values in 'i'.
3. The resulting `sparse_coo_tensor_2` will have a size that is large enough to accommodate the maximum index values. **In this case, the inferred size will be [3, 3] because the maximum indices are 2 along both dimensions.**

3.9 TORCH.NUMEL

```
1  # TORCH.NUMEL
2  tensor_c = torch.randn(1,100)
3  torch.numel(tensor_c)
```

100

Using this function we can return the number of elements in the certain tensor. For an example `tensor_c` was created using `rand` function which can contain 100 random elements. `torch.numel` function returns the number of elements in the `tensor_c`.

3.10 Elementwise Exponential

This function returns the elementwise exponential values of each element.

```
1  #Elementwise exponentiation
2  elementwise_exp_result = torch.exp(tensor_a.float())
3  print(f'Elementwise Exponential of Tensor_a is :',elementwise_exp_result )
```

Elementwise Exponential of Tensor_a is : tensor([[2.7183, 7.3891],
[20.0855, 54.5981]])

3.11 Concatenation

This function concatenates the tensors along with specified dimensions.

```

1  # Concatenation along a dimension
2  concatenation_result = torch.cat((tensor_a, tensor_b), dim=0)
3  print(f'Concatenation along a dimension : ', concatenation_result )

Concatenation along a dimension : tensor([[1, 2],
      [3, 4],
      [5, 6],
      [7, 8]])

```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{matrix} & 1 & 2 \\ 3 & 3 & 4 \\ 5 & 5 & 6 \\ 7 & 7 & 8 \end{matrix}$$

3.12 Reshape the Tensor

This function reshapes the tensor in to new shape.

```

1  # Reshape tensor
2  reshaped_result = tensor_a.view(-1)
3  print(f'reshaped_result : ', reshaped_result )

reshaped_result : tensor([1, 2, 3, 4])

```

3.13 Elementwise less than and Greater than

These functions subsequently perform elementwise greater than and elementwise less than comparison. As a result, they output True or False.

```

1  #Elementwise greater than
2  elementwise_greater_than_result = torch.gt(tensor_a, tensor_b)
3
4  #Elementwise less than
5  elementwise_less_than_result = torch.lt(tensor_a, tensor_b)
6  print("\nElementwise Greater Than Result:")
7  print(elementwise_greater_than_result)
8  print("\nElementwise Less Than Result:")
9  print(elementwise_less_than_result)

```

```

Elementwise Greater Than Result:
tensor([[False, False],
        [False, False]])

Elementwise Less Than Result:
tensor([[True, True],
        [True, True]])

```

3.14 Calculate the determinate of a Matrix.

First recreated the tensor_a and tensor_b with data type float32.

```

[47] 1  tensor_a = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
      2  tensor_b = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

```

```

1  #Matrix determinant
2  matrix_determinant = torch.det(tensor_a)
3  print("Matrix Determinant:")
4  print(matrix_determinant)
5

```

```

Matrix Determinant:
tensor(-2.)

```

This function calculates the determinant of the tensor_a and save it into matrix_determinant tensor.

3.15 TORCH.FULL

This function creates a tensor for the given size and filled with the given value. Tensor's data type will infer from the fill value.

```
1 #TORCH.FULL
2 torch.full((2,2),5)

tensor([[5, 5],
        [5, 5]])
```

3.16 TORCH.MASKED_SELECTED

```
1 #TORCH.MASKED_SELECT
2 tensor_d = torch.rand(2,2)
3 print(tensor_d, '\n')
4 mask = tensor_d.ge(0.5)
5 print(mask, '\n')
6 selected_values = torch.masked_select(tensor_d, mask)
7 print(selected_values)

tensor([[0.2868, 0.0913],
        [0.5726, 0.6436]])

tensor([[False, False],
        [ True,  True]])

tensor([0.5726, 0.6436])
```

To get an idea about this function, first 2X2 tensor was created and named as `tensor_d`. Then create a mask to get the values greater than 0.5. These masks contained Boolean format data. Then a selected value variable was created and using the function `torch.masked_selected()` all the values which are greater than 0.5 were stored in the tensor.

3.17 Batch Matrix Multiplication

```

1 # Batch matrix multiplication
2 batched_matrix_a = torch.rand(3, 2, 2)
3 batched_matrix_b = torch.rand(3, 2, 2)
4 print(batched_matrix_a, '\n', batched_matrix_b, '\n')
5 batched_matrix_multiply_result = torch.bmm(batched_matrix_a, batched_matrix_b)
6 print("\nBatch Matrix Multiplication Result:")
7 print(batched_matrix_multiply_result)

```

```

tensor([[[0.8176, 0.9138],
         [0.2785, 0.1611]],

        [[0.9226, 0.8993],
         [0.0601, 0.2610]],

        [[0.2041, 0.5310],
         [0.8635, 0.5005]]])
tensor([[[0.7942, 0.3209],
         [0.1028, 0.8022]],

        [[0.4398, 0.6017],
         [0.8370, 0.5187]],

        [[0.0945, 0.6098],
         [0.7600, 0.8448]]])

Batch Matrix Multiplication Result:
tensor([[[0.7432, 0.9955],
         [0.2377, 0.2186]],

        [[1.1584, 1.0215],
         [0.2449, 0.1715]],

        [[0.4228, 0.5730],
         [0.4620, 0.9493]]])

```

Two tensors were created calling `batch_matrix_a` and `batch_matrix_b`. each tensor containing 3 matrices, size of 2x2. **torch.bmm** perform batch matrix multiplication between two batches. Each 2x2 matrix in `batched_matrix_a` multiply with corresponding 2x2 matrices in `batched_matrix_b`. The result matrix named `batch_matrix_multiplication_result` containing 3, 2x2 matrices.

3.18 TORCH.VSPLIT

```
1 # TORCH.VSPLIT
2 # Create a 4x4 tensor
3 t = torch.arange(16.0).reshape(4, 4)
4 print(t)
5
6 # Split the tensor into two along the rows
7 split_result_1 = torch.vsplit(t, 2)
8 print(split_result_1)
9
10 # Split the tensor into three along the rows with specified indices
11 split_result_2 = torch.vsplit(t, [3, 6])
12 print(split_result_2)
13
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
(tensor([[0., 1., 2., 3.],
        [4., 5., 6., 7.]]), tensor([[ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
(tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]]), tensor([[12., 13., 14., 15.]]), tensor([], size=(0, 4)))
```

First created a 4x4 tensor which contained values ranging from 0 to 15. **torch.vsplit(t,2)** function splits the tensor t in to 2 tensors along with the rows and save it as split_result_1. Then second variable named split_result_2 created and using the **torch.vsplit(t,[3,6])** function, it split in to 3 tensors along with rows and split points are specified as 3 and 6.

3.19 Repeat Tensor Along with the Dimension

```
1 #Repeat tensor along a dimension
2 repeated_tensor = tensor_a.repeat(2, 3)
3 print("\nRepeated Tensor:")
4 print(repeated_tensor)
```

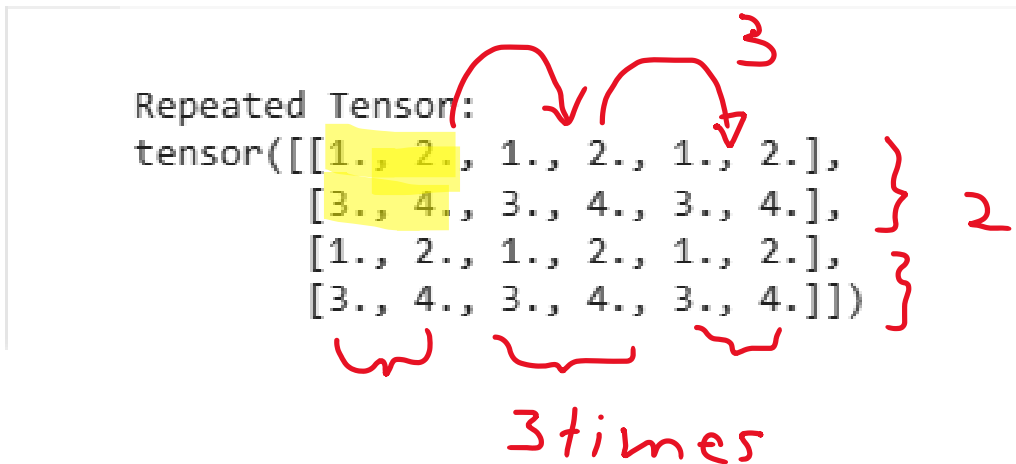
```
Repeated Tensor:
tensor([[1., 2., 1., 2., 1., 2.],
        [3., 4., 3., 4., 3., 4.],
        [1., 2., 1., 2., 1., 2.],
        [3., 4., 3., 4., 3., 4.]])
```

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ Tensor_a is 2x2 tensor.

The '**tensor_a.repeat(2, 3)**' will be perform as bellow,

1. The first dimension (rows) is repeated 2 times, and the second dimension (columns) is repeated 3 times.

- The resulting tensor has a shape of (4, 6) because 2 times 2 along the rows and 3 times 2 along the columns.



3.20 Cumulative Product along the Dimension

```
1 tensor_e = torch.tensor([[2,3,4],[5,2,1]])
2
3 # Cumulative product along a dimension
4 cumulative_product_result = torch.cumprod(tensor_e, dim=1)
5 print("\nCumulative Product Result:")
6 print(cumulative_product_result)
```

```
Cumulative Product Result:
tensor([[ 2,  6, 24],
        [ 5, 10, 10]])
```

- Along the specified dimension ($\text{dim}=1$), the cumulative product is computed for each row.
- For the first row $[2, 3, 4]$, the cumulative product along the columns is $[2, 2*3, 2*3*4]$ resulting in $[2, 6, 24]$.
- For the second row $[5, 2, 1]$, the cumulative product along the columns is $[5, 5*2, 5*2*1]$ resulting in $[5, 10, 10]$.

3.21 TORCH.PERMUTE.

```
1 # TORCH.PERMUTE
2 x = torch.randn(2, 3, 5)
3 print(x)
4 print(x.size(), '\n')
5
6 torch.permute(x, (2, 0, 1))
```

tensor([[[1.5880, 0.7639, 0.5755, 0.2814, -0.2978],
 [0.5198, -0.9649, -0.7114, -0.7560, 0.8767],
 [-0.3643, 0.2914, 1.1468, 0.7619, -0.7350]],
 [[0.2616, 1.0866, 0.5497, 0.8726, -0.6225],
 [0.6467, 0.6140, -0.2402, 0.7382, -0.1878],
 [0.8570, -0.4697, 0.6649, -0.3207, 1.0428]]])
torch.Size([2, 3, 5])

tensor([[[1.5880, 0.5198, -0.3643],
 [0.2616, 0.6467, 0.8570]],
 [[0.7639, -0.9649, 0.2914],
 [1.0866, 0.6140, -0.4697]],
 [[0.5755, -0.7114, 1.1468],
 [0.5497, -0.2402, 0.6649]],
 [[0.2814, -0.7560, 0.7619],
 [0.8726, 0.7382, -0.3207]],
 [[-0.2978, 0.8767, -0.7350],
 [-0.6225, -0.1878, 1.0428]]])

torch.permute(x, (2, 0, 1)).size()

This uses **torch.permute** to permute the dimensions of the tensor. The argument **(2, 0, 1)** specifies the new order of dimensions. The result is a tensor with dimensions **[5, 2, 3]**.

So, the original tensor **x** has dimensions **[2, 3, 5]**, and after permuting the dimensions using **(2, 0, 1)**, the resulting tensor has dimensions **[5, 2, 3]**. The order of dimensions is changed accordingly.

3.22 TORCH.WHERE

- **torch.where** is used for element-wise conditional selection in PyTorch.
- The function takes three arguments: **condition**, **x**, and **y**.
 - **condition**: A boolean tensor of the same shape as **x** and **y**. The values of **x** are selected where **condition** is **True**, and the values of **y** are selected where **condition** is **False**.
 - **x**: The tensor whose elements are selected where the condition is **True**.
 - **y**: The tensor whose elements are selected where the condition is **False**.
 - The result tensor has the same shape as **condition**, **x**, and **y**

```
1 # Define a condition and two tensors
2 condition = torch.tensor([[True, False], [False, True]])
3 x = torch.tensor([[1, 2], [3, 4]])
4 y = torch.tensor([[5, 6], [7, 8]])
5
6 # Use torch.where to select elements based on the condition
7 result = torch.where(condition, x, y)
8 print(result)
9
```

```
tensor([[1, 6],
        [7, 4]])
```

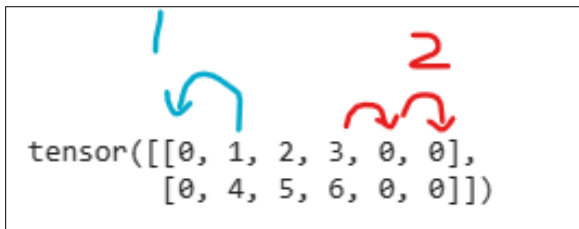
- The **condition** tensor has the values **[[True, False], [False, True]]**.
- The elements of **x** are selected where **condition** is **True**, and the elements of **y** are selected where **condition** is **False**.
- The resulting tensor is **[[1, 6], [7, 4]]**, where **[1, 6]** corresponds to selecting elements from **x**, and **[7, 4]** corresponds to selecting elements from **y** based on the condition.

3.23 TORCH.NN.FUNCTIONAL.PAD

```
1 import torch.nn.functional as F
2
3 # Define a tensor
4 input_tensor = torch.tensor([[1, 2, 3],
5                               [4, 5, 6]])
6
7 # Apply zero-padding using F.pad
8 padded_tensor = F.pad(input_tensor, (1, 2))
9 print(padded_tensor)
```

tensor([[0, 1, 2, 3, 0, 0],
 [0, 4, 5, 6, 0, 0]])

- **torch.nn.functional.pad** takes two arguments: the input tensor and a tuple specifying the padding for each dimension.
- In the example, the input tensor is a 2x3 tensor.
- The tuple **(1, 2)** indicates padding of 1 element on the left and 2 elements on the right along the second dimension.
- The resulting padded tensor will have a shape of **(2, 3 + 1 + 2) = (2, 6)**
- Zeros are added on the left and right sides of each row to achieve the specified padding.
- The first and last columns are padded with zeros according to the specified padding configuration.



tensor([[0, 1, 2, 3, 0, 0],
 [0, 4, 5, 6, 0, 0]])

3.24 TORCH.ARGMAX

- **torch.argmax** returns the indices of the maximum values in a tensor.
- By default, it returns a single index corresponding to the flattened array of the input tensor. This is useful when you want the index of the overall maximum value.

```
1 # Create a tensor
2 input_tensor = torch.tensor([[3, 1, 4], [1, 5, 9]])
3
4 # Find the indices of the maximum values along a specified axis (default: last axis)
5 max_indices = torch.argmax(input_tensor)
6 print(max_indices)
7
```

tensor(5)

3.25 TORCH.AUTOGRAD.GRAD

```
1 # Define a tensor and create a computation graph
2 x = torch.tensor(2.0, requires_grad=True)
3 y = x ** 2
4 z = y + 3
5
6 # Compute the gradient of z with respect to x using torch.autograd.grad
7 gradient = torch.autograd.grad(z, x)
8 print("Gradient:", gradient[0].item())
9
```

Gradient: 4.0

- **torch.autograd.grad** is used to compute the gradient of a scalar-valued tensor (**z** in this case) with respect to other tensors (**x** in this case).
- The input to **torch.autograd.grad** is the target tensor (**z**) and the source tensor (**x**).
- The result is a tuple containing the gradient with respect to each source tensor. In this example, **gradient[0]** contains the gradient with respect to **x**.
- The **requires_grad=True** flag is set on **x** to track operations for automatic differentiation.
- The gradient of $z = x^2 + 3$ with respect to **x** is $2 * x = 2 * 2 = 4.0$

References

[1]

“torch — PyTorch 1.12 documentation,” *pytorch.org*. <https://pytorch.org/docs/stable/torch.html>

[2]

“Tensor Math Operations - Deep Learning with PyTorch 4,” *www.youtube.com*.
<https://www.youtube.com/watch?v=Ta3z9vZaoMc&t=167s> (accessed Mar. 08, 2024).

Appendix

[Colab Code](#)