nDimensional-tensorsVsNumpy

November 16, 2020

0.0.1 Pytorch Foundation, The Tensors

[1]:
```
# libraries
import numpy as np
import torch
import pandas as pd
import datetime
```

[2]:
```
epoch = datetime.datetime.now()
epoch
```

[2]:
```
datetime.datetime(2020, 11, 15, 23, 21, 24, 739036)
```

[3]:
```
epoch.strftime("%B %d, %Y")
```

[3]:
```
'November 15, 2020'
```

[4]:
```
# version check
torch.__version__
```

[4]:
```
'1.6.0'
```

0.0.2 Table of Contents

- Types and Shapes
- Indexing and slicing
- Tensor Functions
- Tensor Operations
- Broadcasting

Types and Shapes

[5]:
```
# integer tensor
tensor_arr = torch.tensor([1,2,3,4,5])
# list parsed into the torch.tensor() function which are then converted to a␣
↪longTensor
```

```
[6]: #dtype
tensor_arr.dtype
```

```
[6]: torch.int64
```

```
[7]: # type of tensor
tensor_arr.type()
```

```
[7]: 'torch.LongTensor'
```

```
[8]: tensor_arr1 = torch.tensor([0.0,1.1,1.2,3.2])
```

```
[9]: #dtype
tensor_arr1.dtype
```

```
[9]: torch.float32
```

```
[10]: # type of tensor
tensor_arr1.type()
```

```
[10]: 'torch.FloatTensor'
```

```
[11]: # TYPE CASTING a Tensor
new_tensor = torch.tensor([1,2,3,8,6,4])
```

```
[12]: new_tensor.dtype
```

```
[12]: torch.int64
```

```
[13]: # typecasting
typecasted_tensor = new_tensor.type(torch.FloatTensor)
```

```
[14]: typecasted_tensor.dtype
```

```
[14]: torch.float32
```

```
[15]: typecasted_tensor
```

```
[15]: tensor([1., 2., 3., 8., 6., 4.])
```

Note, it's often preferred to leave the float-type feature data (tensor) with as much Precision as possible for faster computation and no loss of information

0.0.3 Numpy to Tensor and Back

```
[16]: nparr  = np.arange(0,1,0.01)
```

[17]:
```
nparr
```

[17]:
```
array([0.  , 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1 ,
       0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 , 0.21,
       0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 , 0.31, 0.32,
       0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4 , 0.41, 0.42, 0.43,
       0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5 , 0.51, 0.52, 0.53, 0.54,
       0.55, 0.56, 0.57, 0.58, 0.59, 0.6 , 0.61, 0.62, 0.63, 0.64, 0.65,
       0.66, 0.67, 0.68, 0.69, 0.7 , 0.71, 0.72, 0.73, 0.74, 0.75, 0.76,
       0.77, 0.78, 0.79, 0.8 , 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87,
       0.88, 0.89, 0.9 , 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98,
       0.99])
```

[18]:
```
type(nparr)
```

[18]:
```
numpy.ndarray
```

[19]:
```
tensor_from_np = torch.from_numpy(nparr)
```

[20]:
```
tensor_from_np.type
```

[20]:
```
<function Tensor.type>
```

[21]:
```
# back to numpy
backtonp = tensor_from_np.numpy()
```

[22]:
```
type(backtonp)
```

[22]:
```
numpy.ndarray
```

Tensor from a Pandas Series and Back

[23]:
```
pandasSeries = pd.Series([1,2,3,4,5,8,9])
```

[24]:
```
pandasSeries
```

[24]:
```
0    1
1    2
2    3
3    4
4    5
5    8
6    9
dtype: int64
```

[25]:
```
type(pandasSeries)
```

[25]: pandas.core.series.Series

[26]:
```
tensor_from_pd = torch.from_numpy(pandasSeries.values)
# Since, Every pandas Series is a Numpy array, and Every Dataframe column is a␣
↪Pandas Series
```

[27]:
```
tensor_from_pd
```

[27]: tensor([1, 2, 3, 4, 5, 8, 9])

Tensor View The tensor_obj.view(row, column) is used for reshaping a tensor object.

[28]:
```
tensorview = tensor_from_np[:10]
```

[29]:
```
tensorview
```

[29]:
```
tensor([0.0000, 0.0100, 0.0200, 0.0300, 0.0400, 0.0500, 0.0600, 0.0700, 0.0800,
        0.0900], dtype=torch.float64)
```

[30]:
```
tensorview.size()
```

[30]: torch.Size([10])

[31]:
```
tensorview.view(5,2) # we can view a normal, flattened tensor in a 2d form,
# Note, here (row,col) are the multiples of the size of the tensor
```

[31]:
```
tensor([[0.0000, 0.0100],
        [0.0200, 0.0300],
        [0.0400, 0.0500],
        [0.0600, 0.0700],
        [0.0800, 0.0900]], dtype=torch.float64)
```

[32]:
```
tensorview.view(5,-1) # if we are unsure of the 2nd dimension parameterizing -1␣
↪will
# intuitively know the second dimension providing the first dimension should be␣
↪correct multiple.
```

[32]:
```
tensor([[0.0000, 0.0100],
        [0.0200, 0.0300],
        [0.0400, 0.0500],
        [0.0600, 0.0700],
        [0.0800, 0.0900]], dtype=torch.float64)
```

[33]:
```
# Retrieving a Value from Tensor
```

[34]:
```
tensor_arr1
```

[34]: tensor([0.0000, 1.1000, 1.2000, 3.2000])

[35]:
```
tensor_arr1[0].item() # retrieves item as a standard python number, only works␣
↪for one element
```

[35]: 0.0

[36]:
```
# tolist
tensor_arr1.tolist() # to a standard python list
```

[36]: [0.0, 1.100000023841858, 1.2000000476837158, 3.200000047683716]

0.0.4 Indexing and Slicing

[37]:
```
index_tensor = torch.tensor([0, 1, 2, 3, 4])
print("The value on index 0:",index_tensor[0])
print("The value on index 1:",index_tensor[1])
print("The value on index 2:",index_tensor[2])
print("The value on index 3:",index_tensor[3])
print("The value on index 4:",index_tensor[4])
```

```
The value on index 0: tensor(0)
The value on index 1: tensor(1)
The value on index 2: tensor(2)
The value on index 3: tensor(3)
The value on index 4: tensor(4)
```

[38]:
```
index_tensor[0:3] # slicing a Tensor, this is a subset of the original tensor
```

[38]: tensor([0, 1, 2])

[39]:
```
len(index_tensor)
```

[39]: 5

[40]:
```
assert(index_tensor[5]) # out of bounds, since the tensor is 0 till 4 at the␣
↪index
```

```
␣
↪---------------------------------------------------------------------------

    IndexError                                Traceback (most recent call␣
↪last)

    <ipython-input-40-c22ef94243c0> in <module>
```

```
----> 1 assert(index_tensor[5]) # out of bounds, since the tensor is 0 till␣
↪4 at the index


    IndexError: index 5 is out of bounds for dimension 0 with size 5
```

[ ]:
```
#. assigning values at indexes
tensor_sample = torch.tensor([20, 1, 2, 3, 4])
```

[ ]:
```
print("Inital value on index 0:", tensor_sample[0])
tensor_sample[0] = 100
print("Modified tensor:", tensor_sample)
```

0.0.5 Tensor Operations

[60]:
```
np.random.seed(101)
random_tensor = torch.from_numpy(np.random.randint(10,50,100))
```

[61]:
```
random_tensor
```

[61]:
```
tensor([41, 21, 27, 16, 33, 21, 19, 23, 14, 38, 10, 15, 22, 39, 29, 18, 39, 44,
        18, 29, 20, 22, 41, 33, 10, 19, 18, 46, 29, 45, 38, 17, 20, 49, 48, 19,
        28, 17, 49, 25, 10, 22, 27, 21, 25, 43, 39, 34, 46, 29, 45, 40, 20, 49,
        30, 37, 18, 32, 36, 33, 47, 32, 19, 12, 28, 38, 21, 20, 40, 45, 38, 13,
        29, 30, 24, 15, 15, 16, 34, 49, 47, 17, 47, 14, 33, 45, 25, 44, 13, 28,
        23, 13, 47, 39, 32, 31, 31, 27, 33, 40])
```

[62]:
```
random_tensor.type()
```

[62]:
```
'torch.LongTensor'
```

[63]:
```
# MEAN of a Tensor
random_tensor.mean()
# note, tensors doesn't calculate mean on Longtensor i.e the Integer arrays,␣
↪they need to be
# typecasted for that to happen
```

```
        ␣
↪---------------------------------------------------------------------------

    RuntimeError                              Traceback (most recent call␣
↪last)

    <ipython-input-63-bbaca88aaca7> in <module>
      1 # MEAN of a Tensor
```

```
----> 2 random_tensor.mean()
      3 # note, tensors doesn't calculate mean on Longtensor i.e the Integer␣
↪arrays, they need to be
      4 # typecasted for that to happen


    RuntimeError: Can only calculate the mean of floating types. Got Long␣
↪instead.
```

[64]:
```
# typecasting
random_tensorF = random_tensor.type(torch.float32)
```

[65]:
```
random_tensorF.dtype
```

[65]:
```
torch.float32
```

[66]:
```
random_tensorF
```

[66]:
```
tensor([41., 21., 27., 16., 33., 21., 19., 23., 14., 38., 10., 15., 22., 39.,
        29., 18., 39., 44., 18., 29., 20., 22., 41., 33., 10., 19., 18., 46.,
        29., 45., 38., 17., 20., 49., 48., 19., 28., 17., 49., 25., 10., 22.,
        27., 21., 25., 43., 39., 34., 46., 29., 45., 40., 20., 49., 30., 37.,
        18., 32., 36., 33., 47., 32., 19., 12., 28., 38., 21., 20., 40., 45.,
        38., 13., 29., 30., 24., 15., 15., 16., 34., 49., 47., 17., 47., 14.,
        33., 45., 25., 44., 13., 28., 23., 13., 47., 39., 32., 31., 31., 27.,
        33., 40.])
```

[67]:
```
random_tensorF.mean()
```

[67]:
```
tensor(29.3900)
```

[69]:
```
np.random.seed(101)
np.random.randint(10,50,100).mean() # whereas in Numpy it's possible
```

[69]:
```
29.39
```

[71]:
```
# MEDIAN
random_tensorF.median()
```

[71]:
```
tensor(29.)
```

[72]:
```
# standard deviation
random_tensorF.std()
```

[72]:
```
tensor(11.3990)
```

0.0.6 Min & Max

[73]:
```
random_tensor
```

[73]:
```
tensor([41, 21, 27, 16, 33, 21, 19, 23, 14, 38, 10, 15, 22, 39, 29, 18, 39, 44,
        18, 29, 20, 22, 41, 33, 10, 19, 18, 46, 29, 45, 38, 17, 20, 49, 48, 19,
        28, 17, 49, 25, 10, 22, 27, 21, 25, 43, 39, 34, 46, 29, 45, 40, 20, 49,
        30, 37, 18, 32, 36, 33, 47, 32, 19, 12, 28, 38, 21, 20, 40, 45, 38, 13,
        29, 30, 24, 15, 15, 16, 34, 49, 47, 17, 47, 14, 33, 45, 25, 44, 13, 28,
        23, 13, 47, 39, 32, 31, 31, 27, 33, 40])
```

[74]:
```
# maximum
random_tensor.max()
```

[74]:
```
tensor(49)
```

[77]:
```
# minimum
random_tensor.min()
```

[77]:
```
tensor(10)
```

0.0.7 Broadcasting

[80]:
```
tensor = torch.from_numpy(np.random.randint(10,99,2))
```

[81]:
```
tensor
```

[81]:
```
tensor([81, 30])
```

[82]:
```
# elementwise addition - Exibit -1
2 + tensor # tensor(vector) + scalar operation ( scalar is broadcasted to every␣
↪vector element)
```

[82]:
```
tensor([83, 32])
```

[84]:
```
# Exibit -2
u = torch.tensor([1, 0])
v = torch.tensor([0, 1])
w = u + v
w
# The result is tensor([1, 1]). The behavior is [1 + 0, 0 + 1].
```

[84]:
```
tensor([1, 1])
```

[87]:
```
# Multiplication  (elementwise)

u = torch.tensor([1, 2])
```

```
v = torch.tensor([3, 2])
w = u * v
w
# The result is simply tensor([3, 4]).
# This result is achieved by multiplying every element in u
# with the corresponding element in the same position v, which is similar to [1␣
↪* 3, 2 * 2].
```

[87]: tensor([3, 4])

[91]:
```
u = torch.tensor([1, 2])
v = torch.tensor([3, 2])
torch.dot(u,v)
# The result is tensor(7). The function is 1 x 3 + 2 x 2 = 7.
```

[91]: tensor(7)

0.0.8 2 Dimensional Tensors

[99]:
```
! ../../epochgen.py ef
```

```
Epoch ~ (2020-11-16 00:12:58.331059)
Decomposed Date ~  November 16, 2020;
```

[100]:
```
twoD_list = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

[101]:
```
tensor2d = torch.tensor(twoD_list)
```

[102]:
```
tensor2d
```

[102]:
```
tensor([[11, 12, 13],
        [21, 22, 23],
        [31, 32, 33]])
```

[103]:
```
# type
tensor2d.type()
```

[103]: 'torch.LongTensor'

[105]:
```
tensor2d.dtype
```

[105]: torch.int64

[108]:
```
# size
tensor2d.size()
```

[108]: torch.Size([3, 3])

[111]:
```
# shape
tensor2d.shape
# tensor object has same outputs for the shape and size, unlike numpy where two␣
↪dimensions
# Row, col are mutiplied to output the size
```

[111]:
```
torch.Size([3, 3])
```

[112]:
```
print("The dimension of tensor2d: ", tensor2d.ndimension())
print("The shape of tensor2d: ", tensor2d.shape)
print("The shape of tensor2d: ", tensor2d.size())
print("The number of elements in tensor2d: ", tensor2d.numel())
```

```
The dimension of tensor2d:  2
The shape of tensor2d:  torch.Size([3, 3])
The shape of tensor2d:  torch.Size([3, 3])
The number of elements in tensor2d:  9
```

[114]:
```
twoD_numpy = tensor2d.numpy()
print("Tensor -> Numpy Array:")
print("The numpy array after converting: ", twoD_numpy)
print("Type after converting: ", twoD_numpy.dtype)

print("================================================")

new_tensor2d = torch.from_numpy(twoD_numpy)
print("Numpy Array -> Tensor:")
print("The tensor after converting:", new_tensor2d)
print("Type after converting: ", new_tensor2d.dtype)
```

```
Tensor -> Numpy Array:
The numpy array after converting:  [[11 12 13]
 [21 22 23]
 [31 32 33]]
Type after converting:  int64
================================================
Numpy Array -> Tensor:
The tensor after converting: tensor([[11, 12, 13],
        [21, 22, 23],
        [31, 32, 33]])
Type after converting:  torch.int64
```

[116]:
```
# Dataframe to tensor
df = pd.DataFrame({'a':[11,21,31],'b':[12,22,312]})
df
```

[116]:
```
     a    b
0   11   12
1   21   22
2   31  312
```

[117]:
```
print("Pandas Dataframe to numpy: ", df.values)
print("Type BEFORE converting: ", df.values.dtype)

print("================================================")

new_tensor = torch.from_numpy(df.values)
print("Tensor AFTER converting: ", new_tensor)
print("Type AFTER converting: ", new_tensor.dtype)
```

```
Pandas Dataframe to numpy:  [[ 11  12]
 [ 21  22]
 [ 31 312]]
Type BEFORE converting:  int64
================================================
Tensor AFTER converting:  tensor([[ 11,  12],
        [ 21,  22],
        [ 31, 312]])
Type AFTER converting:  torch.int64
```

0.0.9 2 Dimensional Indexing

[119]:
```
tensor_example = torch.tensor([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
print("What is the value on 2nd-row 3rd-column? ", tensor_example[1, 2])
print("What is the value on 2nd-row 3rd-column? ", tensor_example[1][2])
```

```
What is the value on 2nd-row 3rd-column?  tensor(23)
What is the value on 2nd-row 3rd-column?  tensor(23)
```

[121]:
```
tensor_example
```

[121]:
```
tensor([[11, 12, 13],
        [21, 22, 23],
        [31, 32, 33]])
```

[123]:
```
# first row, 2 columns ?
tensor_example[0,0:2]
```

[123]:
```
tensor([11, 12])
```

[126]:
```
# last column,2 rows?
tensor_example[1:,2].reshape(-1,1)
```

[126]: tensor([[23],
        [33]])

0.0.10 2 Dimensional Operations

[127]:
```
# Calculate [[1, 0], [0, 1]] + [[2, 1], [1, 2]]

X = torch.tensor([[1, 0],[0, 1]])
Y = torch.tensor([[2, 1],[1, 2]])
X_plus_Y = X + Y
print("The result of X + Y: ", X_plus_Y)
```

```
The result of X + Y:  tensor([[3, 1],
        [1, 3]])
```

[132]:
```
# Scalar Multiplication
print(Y)
twoy = 2 * Y
twoy
```

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

[132]: tensor([[4, 2],
        [2, 4]])

0.0.11 Hadamard Product (elementwise)

[133]:
```
X = torch.tensor([[1, 0], [0, 1]])
Y = torch.tensor([[2, 1], [1, 2]])
X_times_Y = X * Y
print("The result of X * Y: ", X_times_Y)
```

```
The result of X * Y:  tensor([[2, 0],
        [0, 2]])
```

0.0.12 Matrix Multiplication

[139]:
```
# Calculate [[0, 1, 1], [1, 0, 1]] * [[1, 1], [1, 1], [-1, 1]]

A = torch.tensor([[0, 1, 1], [1, 0, 1]])
B = torch.tensor([[1, 1], [1, 1], [-1, 1]])
A_times_B = torch.mm(A,B)
print("The result of A * B: ", A_times_B)
```

```
The result of A * B:  tensor([[0, 2],
        [0, 2]])
```

[140]:
```
print('A-shape {}'.format(A.shape))
print('B-shape {}'.format(B.shape))
```

```
A-shape torch.Size([2, 3])
B-shape torch.Size([3, 2])
```

[141]:
```
# trying for unequal dimentsions
A = torch.tensor([[0, 1, 1], [1, 0, 1]])
B = torch.tensor([[1, 1], [1, 1]])
A_times_B = torch.mm(A,B)
# condition, pxq matrix is eligible to multiple with other matrix only if it's␣
↪shape is qxp
```

```
        ␣
↪---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call␣
↪last)

        <ipython-input-141-9a7756553d1e> in <module>
          2 A = torch.tensor([[0, 1, 1], [1, 0, 1]])
          3 B = torch.tensor([[1, 1], [1, 1]])
    ----> 4 A_times_B = torch.mm(A,B)
          5 # condition, pxq matrix is eligible to multiple with other matrix␣
↪only if it's shape is qxp


        RuntimeError: size mismatch, m1: [2 x 3], m2: [2 x 2] at ../aten/src/TH/
↪generic/THTensorMath.cpp:41
```

[ ]:
```

```