programs_2

June 10, 2023

[2]:

```
import random
import numpy as np
import tensorflow as tf
```

[2]:

```
# A tuple is only hashable if all of its items are hashable as well.
x = (1,[2,3])
y = (1,2,3)
print(type(x))
print(hash(y))
print(hash(x))
```

```
<class 'tuple'>
2528502973977326415
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5612\411819356.py in <module>
      4 print(type(x))
      5 print(hash(y))
----> 6 print(hash(x))

TypeError: unhashable type: 'list'
```

[ ]:

```
Write the code to get the count of row for each category in the dataframes.
```

Write the code for proportional sampling.

[4]:

```
a = [[1, 2, 3, 10], [4, 5, 6, 11], [7, 8, 9, 12]]
a = np.array(a)
print(a, a.shape)
print(a[:,:-1])
print(a[:-2,:])
```

```
[[ 1  2  3 10]
 [ 4  5  6 11]
 [ 7  8  9 12]] (3, 4)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[ 1  2  3 10]]
```

[8]:

```
a=dict()
a[('a','b')] = 0
b = 0
a[(a,b)] = 1
print(a)

# TypeError: unhashable type: 'dict' =>
# You're trying to use a dict as a key to another dict or in a set.
# That does not work because the keys have to be hashable.
# As a general rule, only immutable objects (strings, integers, floats,␣
 ↪frozensets, tuples of immutables) are hashable (though exceptions are␣
 ↪possible).
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6268\262450242.py in <module>
      2 a[('a','b')] = 0
      3 b = 0
----> 4 a[(a,b)] = 1
      5 print(a)

TypeError: unhashable type: 'dict'
```

[10]:

```
a=b=dict()
a[('a','b')] = 0
a[a] = 1
print(a)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6268\3271372320.py in <module>
      1 a=b=dict()
      2 a[('a','b')] = 0
----> 3 a[a] = 1
      4 print(a)

TypeError: unhashable type: 'dict'
```

[18]:

```
a = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
print(np.array(a), np.array(a).shape)

# compute the arithmetic mean along the specified axis. along the specified␣
 ↪axis not in the specified axis.
```

```
np.mean(a,axis=1)
```

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

```
[18]: array([2., 5., 8.])
```

```
[24]: # part 1
x = np.random.randn(2,4,5)
x[0][0][0]=x[0][0][1]=x[0][0][2]=x[0][0][3]=x[0][0][4]=np.nan
print(x)
print(x.shape)
print(tf.math.is_nan(x).numpy())
```

```
[[[        nan         nan         nan         nan         nan]
  [-0.38741481  0.88251336 -2.97126026 -0.68630933  0.80255069]
  [-0.31116595 -0.28278029  1.27927538 -0.24212618  1.00564663]
  [-0.87610649  1.26589328  0.1419096   0.52338669  2.40479413]]

 [[-0.18900162  0.12356675 -1.77809813 -0.15425863  0.88222628]
  [-0.81170921  0.09287636 -1.04551404 -1.28765236 -1.21925178]
  [-0.14906551  0.37176164  0.39364237 -0.38070733  0.35246171]
  [-1.86615688 -0.40064444 -1.10919396  0.23376485  0.27158802]]]
(2, 4, 5)
[[[ True  True  True  True  True]
  [False False False False False]
  [False False False False False]
  [False False False False False]]

 [[False False False False False]
  [False False False False False]
  [False False False False False]
  [False False False False False]]]
```

```
[22]: # part 2
#z = tf.reduce_all(tf.math.is_nan(x), axis=[-2,-1])
z = tf.reduce_all(tf.math.is_nan(x), axis=[-1])# axis => the dimensions to␣
 ↪reduce.
z.numpy()
```

```
[22]: array([[ True, False, False, False],
       [False, False, False, False]])
```

```
[33]: a = [[3, 4, 5],[6, 7, 8],[9, 10, 11]]
b = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

```python
print(np.array(a), np.array(a).shape, '\n\n' ,np.array(b), np.array(b).shape, '\n\n' ,)

# join a sequence of arrays along a new axis.
c = np.stack( (a,b), axis= 0)
print(c, c.shape, '\n\n' ,)

c_1 = np.stack( (a,b), axis= 1)
print(c_1, c_1.shape, '\n\n' ,)

d = np.vstack( (a,b),)
print(d, d.shape, '\n\n' ,)
```

```
[[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]] (3, 3)

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


[[[ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]]

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


[[[ 3  4  5]
  [ 1  2  3]]

 [[ 6  7  8]
  [ 4  5  6]]

 [[ 9 10 11]
  [ 7  8  9]]] (3, 2, 3)


[[ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]] (6, 3)
```

[ ]:

```

```

[9]:

```
# part 1
x = tf.reshape(tf.range(24), (2,3,4))
print(x.numpy())

print(x.shape)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
(2, 3, 4)
```

[11]:

```
# part 2
p, q, r, s = tf.unstack(x, axis=-1)
print(p, q, r)
# p.shape.as_list()
```

```
tf.Tensor(
[[ 0  4  8]
 [12 16 20]], shape=(2, 3), dtype=int32) tf.Tensor(
[[ 1  5  9]
 [13 17 21]], shape=(2, 3), dtype=int32) tf.Tensor(
[[ 2  6 10]
 [14 18 22]], shape=(2, 3), dtype=int32)
```

[14]:

```
x = tf.constant([1, 4])
y = tf.constant([2, 5])
z = tf.constant([3, 6])
print(tf.stack([x, y, z]))
print(tf.stack([x, y, z], axis=-1))
```

```
tf.Tensor(
[[1 4]
 [2 5]
 [3 6]], shape=(3, 2), dtype=int32)
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

[15]:

```
tf.constant([10,20,12,125,55])[...,None]
```

[15]:

```
<tf.Tensor: shape=(5, 1), dtype=int32, numpy=
array([[ 10],
       [ 20],
       [ 12],
       [125],
       [ 55]], dtype=int32)>
```

[16]:

```
tensor = [0, 0, 0, 0, 0, 0, 0, 0]    # tf.rank(tensor) == 1
indices = [[1], [3], [4], [7]]       # num_updates == 4, index_depth == 1
updates = [9, 10, 11, 12]            # num_updates == 4
print(tf.tensor_scatter_nd_update(tensor, indices, updates))
```

```
tf.Tensor([ 0  9  0 10 11  0  0 12], shape=(8,), dtype=int32)
```

[51]:

```
scale  = (0.7,0.9)
# tf.random.uniform(shape, minval=0, maxval=None, dtype=tf.dtypes.float32)
print(tf.random.uniform((),scale))# 1st minval is 0.7, and 2nd minval is 0.9.
print(tf.random.uniform((),*scale))# minval=0.7, maxval=0.9,
```

```
tf.Tensor([0.752126  0.9173753], shape=(2,), dtype=float32)
tf.Tensor(0.87671584, shape=(), dtype=float32)
```

0.0.1 3d matrix multiplication with 2d

[3]:

```
x = tf.convert_to_tensor(np.random.randn(2,4,2), dtype=tf.float32)
shear_mat = tf.identity([
    [1.,0.93],
    [0,1.]
])
# shear_mat.shape => (2, 2)
print(x, shear_mat)
y = x@shear_mat
print(y)
```

```
tf.Tensor(
[[[ 0.15535602  0.12185005]
  [ 0.0398068   0.55404997]
  [-0.22367506 -1.4742328 ]
  [-1.0462142  -0.42665246]]

 [[-0.05041332  1.1294501 ]
  [-0.7218814  -0.5789553 ]
  [-1.0697414   0.26683167]
  [-0.02007808 -0.71133935]]], shape=(2, 4, 2), dtype=float32) tf.Tensor(
[[1.   0.93]
 [0.   1.  ]], shape=(2, 2), dtype=float32)
tf.Tensor(
[[[ 0.15535602  0.26633114]
```

```
  [ 0.0398068   0.5910703 ]
  [-0.22367506 -1.6822506 ]
  [-1.0462142  -1.3996317 ]]

 [[-0.05041332  1.0825657 ]
  [-0.7218814  -1.2503049 ]
  [-1.0697414  -0.7280278 ]
  [-0.02007808 -0.73001194]]], shape=(2, 4, 2), dtype=float32)
```

[10]:
```
tf.where([True, False, False, True],
         [1, 2, 3, 1],
         [100, 200, 300, 400]).numpy()
```

[10]:
```
array([  1, 200, 300,   1], dtype=int32)
```

[16]:
```
mask_offset_x = tf.constant([0.84])
mask_size = tf.constant([0.33])
x = tf.convert_to_tensor(np.random.randn(2,4,2), dtype=tf.float32)
print(x)
print((mask_offset_x<x[...,0]))
print((x[...,0] < mask_offset_x + mask_size))
mask_x = (mask_offset_x<x[...,0]) & (x[...,0] < mask_offset_x + mask_size)
# mask_x.shape => (2, 4), mask_x.shape => (2, 4, 1)
print(mask_x[...,None])
x = tf.where(mask_x[...,None], float('nan'), x)
print(x)
```

```
tf.Tensor(
[[[-9.2519158e-01 -1.8907794e-03]
  [ 6.2423635e-01 -8.7962878e-01]
  [ 8.7764792e-02  2.0184629e+00]
  [-1.0359342e+00  4.2211890e-01]]

 [[ 1.5514775e-01 -1.1610771e-01]
  [ 8.4216988e-01  2.7054015e-01]
  [ 5.0838810e-01  5.1353163e-01]
  [-1.3255783e+00  1.1715494e+00]]], shape=(2, 4, 2), dtype=float32)
tf.Tensor(
[[False False False False]
 [False  True False False]], shape=(2, 4), dtype=bool)
tf.Tensor(
[[ True  True  True  True]
 [ True  True  True  True]], shape=(2, 4), dtype=bool)
tf.Tensor(
[[[False]
  [False]
  [False]
  [False]]
 [[False]
  [ True]
  [False]
  [False]]], shape=(2, 4, 1), dtype=bool)
tf.Tensor(
[[[-9.2519158e-01 -1.8907794e-03]
  [ 6.2423635e-01 -8.7962878e-01]
  [ 8.7764792e-02  2.0184629e+00]
  [-1.0359342e+00  4.2211890e-01]]

 [[ 1.5514775e-01 -1.1610771e-01]
  [           nan            nan]
  [ 5.0838810e-01  5.1353163e-01]
  [-1.3255783e+00  1.1715494e+00]]], shape=(2, 4, 2), dtype=float32)
```

[6]:

```
# tf.pad ======>
t = tf.constant([[1, 2, 3], [4, 5, 6]])
print(t.numpy(), t.shape, '\n\n')

paddings = tf.constant([[1, 1,], [2, 2]])
# [1, 1] => add 1 value before and 1 value after to the dimension 0 (axis=0).
# [2, 2] => add 2 value before and 2 value after to the dimension 1 (axis=1).

print(paddings.numpy(), paddings.shape, '\n\n')
# 'constant_values' is 0.
# rank of 't' is 2.
z = tf.pad(t, paddings, "CONSTANT")  # [[0, 0, 0, 0, 0, 0, 0],
                                     #  [0, 0, 1, 2, 3, 0, 0],
                                     #  [0, 0, 4, 5, 6, 0, 0],
                                     #  [0, 0, 0, 0, 0, 0, 0]]
print(z.numpy())
```

```
[[1 2 3]
 [4 5 6]] (2, 3)


[[1 1]
 [2 2]] (2, 2)


[[0 0 0 0 0 0 0]
 [0 0 1 2 3 0 0]
 [0 0 4 5 6 0 0]
 [0 0 0 0 0 0 0]]
```

[ ]:

[11]:
```
## tf.cond => (when x and y are equal)
x, y = tf.constant(4, dtype=tf.int32), tf.constant(4, dtype=tf.int32)
z = tf.multiply(x, y)
print(z)
r = tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square(y))
r.numpy()
```

```
tf.Tensor(16, shape=(), dtype=int32)
```

[11]:
```
16
```

[37]:
```
## tf.keras.activations.softmax =>
x=np.random.randint(0, 50, size=(2,3,4,2))
#x=tf.random.normal(shape=(2,3,4,2))
tf_1 = tf.convert_to_tensor(x, dtype=tf.float32)
# apply softmax over 2 classes (present in last dimension) for each 4 rows␣
 ↪(present in 3rd dimension).
# 3rd dimension means axis=2
y=tf.keras.activations.softmax(tf_1, axis=-1)
print(tf_1.numpy())
print(y.numpy())
```

```
[[[[49. 39.]
   [ 1.  5.]
   [16.  6.]
   [11. 24.]]

  [[36.  2.]
   [44. 47.]
   [ 9. 43.]
   [32. 34.]]

  [[44. 45.]
   [32. 39.]
   [15. 41.]
   [46. 45.]]]


 [[[42. 15.]
   [22. 13.]
   [14.  8.]
   [ 5. 21.]]

  [[31.  3.]
   [41.  3.]
   [43.  2.]
   [43. 47.]]
  [[45. 28.]
   [18.  4.]
   [49. 20.]
   [25. 19.]]]]
[[[[9.9995458e-01 4.5397868e-05]
   [1.7986210e-02 9.8201376e-01]
   [9.9995458e-01 4.5397868e-05]
   [2.2603242e-06 9.9999774e-01]]

  [[1.0000000e+00 1.7139085e-15]
   [4.7425874e-02 9.5257413e-01]
   [1.7139085e-15 1.0000000e+00]
   [1.1920292e-01 8.8079703e-01]]

  [[2.6894143e-01 7.3105854e-01]
   [9.1105123e-04 9.9908900e-01]
   [5.1090889e-12 1.0000000e+00]
   [7.3105854e-01 2.6894143e-01]]]


 [[[1.0000000e+00 1.8795287e-12]
   [9.9987662e-01 1.2339458e-04]
   [9.9752742e-01 2.4726233e-03]
   [1.1253516e-07 9.9999988e-01]]

  [[1.0000000e+00 6.9144002e-13]
   [1.0000000e+00 3.1391326e-17]
   [1.0000000e+00 1.5628822e-18]
   [1.7986210e-02 9.8201376e-01]]

  [[1.0000000e+00 4.1399378e-08]
   [9.9999917e-01 8.3152804e-07]
   [1.0000000e+00 2.5436657e-13]
   [9.9752742e-01 2.4726233e-03]]]]
```

[50]:

```
# // => Floor division (take to the nearest lower integer) =>
print((9/-2))
print(9/2)
print("\n")
print((9//-2))
print(9//2)
```

```
-4.5
4.5


-5
4
```

[ ]:

[ ]: