School of InfoComm Technology — Taking IT Higher

Deep Learning

Practical 1 - Tensor Operations

AY2020/21 Semester

Objectives

After completing this practical exercise, students should be able to:

1. [Understand how the data is represented using tensors]
2. [Understand the basics of tensor operations]
3. [Exercise: provide your own examples]

1. Data Representations

All current machine-learning systems use tensors as their basic data structure. A tensor is a container for data, almost always numeric data.

1.1 Scalars (0D tensors)

A tensor that contains only one number.

In [1]:

```
import numpy as np
x = np.array(12)
print('x = \n', x, '\n')
print('The dimension of x is: ', x.ndim)
print('The shape of x is: ', x.shape)
```

```
x =
 12

The dimension of x is:  0
The shape of x is:  ()
```

1.2 Vectors (1D tensors)

An array of numbers.

In [2]:
```
x = np.array([12,3,6,14,25])
print('x = \n', x, '\n')
print('The dimension of x is: ', x.ndim)
print('The shape of x is: ', x.shape)
```

```
x = 
 [12  3  6 14 25] 

The dimension of x is:  1
The shape of x is:  (5,)
```

This vector has five entries and so it is a 5-dimensional vector (5D vector), but a 1D tensor.

1.3 Matrices (2D tensors)

An array of vectors.

In [3]:
```
x = np.array([[5, 78, 2, 34, 0],
              [6, 79, 3, 35, 1],
              [7, 80, 4, 36, 2]])
print('x = \n', x, '\n')
print('The dimension of x is: ', x.ndim)
print('The shape of x is: ', x.shape)
```

```
x = 
 [[ 5 78  2 34  0]
 [ 6 79  3 35  1]
 [ 7 80  4 36  2]] 

The dimension of x is:  2
The shape of x is:  (3, 5)
```

1.4 3D tensors and higher-dimentional tensors

Pack matrices into a new array.

In [4]:
```python
x = np.array([[[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
              [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
              [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]]])
print('x = \n', x)
print('\nThe dimension of x is: ', x.ndim)
print('The shape of x is: ', x.shape)
```

```
x = 
 [[[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]

 [[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]

 [[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]]

The dimension of x is:  3
The shape of x is:  (3, 3, 5)
```

1.5 Key attributes

- Number of axes (rank or dimension): ndim in Numpy
- Shape: how many dimensions the tensor has along each axis
- Data type: dtype in Python. The type of the data contained in the tensor

In [5]:
```python
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

In [6]:
```python
print(train_images.ndim)
```

```
3
```

In [7]:
```
print(train_images.shape)
```

```
(60000, 28, 28)
```

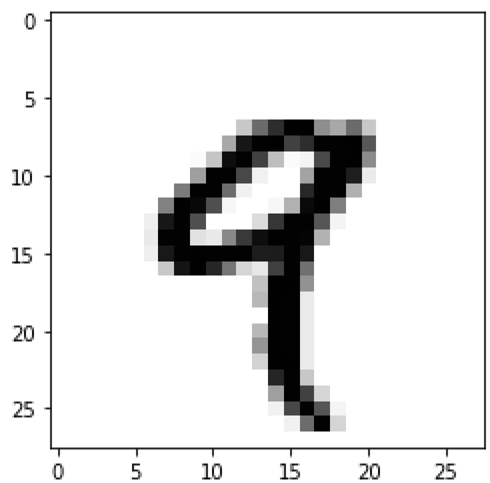
In [8]:
```
print(train_images.dtype)
```

```
uint8
```

So what we have here is a 3D tensor of 8-bit integers. More precisely, it’s an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255. Let’s display the fourth digit in this 3D tensor, using the library Matplotlib.

In [9]:
```
digit = train_images[4]
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
#print(digit)
```



1.6 Tensor Slicing

Selecting specific elements in a tensor

In [10]:

```
#Select digits #10 to #100 (#100 isn't included)
my_slice = train_images[10:100]
print(my_slice.shape)
```

```
(90, 28, 28)
```

In [11]:

```
#Select digits #10 to #100 (#100 isn't included)
my_slice = train_images[10:100, :, :]
print(my_slice.shape)
```

```
(90, 28, 28)
```

In [12]:

```
#Select digits #10 to #100 (#100 isn't included)
my_slice = train_images[10:100, 0:28, 0:28]
print(my_slice.shape)
```

```
(90, 28, 28)
```

In [13]:

```
#Select 14x14 pixels in the bottom-right corner of all images
my_slice = train_images[:, 14:, 14:]
print(my_slice.shape)
```

```
(60000, 14, 14)
```

In [14]:

```
#Crop the images to patches of 14x14 pixels centered in the middle
my_slice = train_images[:, 7:-7, 7:-7]
print(my_slice.shape)
```

```
(60000, 14, 14)
```

Deep-learning models don’t process an entire dataset at once; they break the data into small batches. Here’s one batch of our MNIST digits, with batch size of 128:

In [15]:
```
batch = train_images[:128] # 1st batch
batch = train_images[128:256] # 2nd batch

# the nth batch
n=10
batch = train_images[128 * n:128 * (n + 1)]
print(batch.shape)
```

```
(128, 28, 28)
```

When considering such a batch tensor, the first axis (axis 0) is called the batch axis or batch dimension.

2. Tensor Operations

All transformations learned by deep neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data, e.g. add tensors, multiply tensors and etc.

2.1 Element-wise operations

Operations that are applied independently to each entry in the tensors.

In [16]:
```python
import numpy as np
print('x is: \n', x)
#Element-wise substraction
print("\n==Element-wise substraction==")
print('{:^28}'.format("y = x - 4"))
y = x - 4
print('y is: \n', y)

#Element-wise addition
print("\n==Element-wise addition==")
print('{:^28}'.format("z = x + y"))
z = x + y
print('z is: \n', z)

#Element-wise relu
print("\n==Element-wise relu==")
z2 = np.maximum(z, 0.)
print('z2 is: \n', z2)
```

```
x is: 
 [[[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]

 [[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]

 [[ 5 78  2 34  0]
  [ 6 79  3 35  1]
  [ 7 80  4 36  2]]]

==Element-wise substraction==
         y = x - 4          
y is: 
 [[[ 1 74 -2 30 -4]
  [ 2 75 -1 31 -3]
  [ 3 76  0 32 -2]]

 [[ 1 74 -2 30 -4]
  [ 2 75 -1 31 -3]
  [ 3 76  0 32 -2]]

 [[ 1 74 -2 30 -4]
  [ 2 75 -1 31 -3]
  [ 3 76  0 32 -2]]]
```

```
==Element-wise addition==
         z = x + y
z is:
 [[[  6 152   0  64  -4]
  [  8 154   2  66  -2]
  [ 10 156   4  68   0]]

 [[  6 152   0  64  -4]
  [  8 154   2  66  -2]
  [ 10 156   4  68   0]]

 [[  6 152   0  64  -4]
  [  8 154   2  66  -2]
  [ 10 156   4  68   0]]]

==Element-wise relu==
z2 is:
 [[[  6. 152.   0.  64.   0.]
  [  8. 154.   2.  66.   0.]
  [ 10. 156.   4.  68.   0.]]

 [[  6. 152.   0.  64.   0.]
  [  8. 154.   2.  66.   0.]
  [ 10. 156.   4.  68.   0.]]

 [[  6. 152.   0.  64.   0.]
  [  8. 154.   2.  66.   0.]
  [ 10. 156.   4.  68.   0.]]]
```

2.2 Broadcasting

When the shapes of two tensors being added are different, if there's no ambiguity, the smaller tensor will be broadcasted to match the shape of the larger tensor.

In [17]:
```python
import numpy as np
#Generate random numbers from 0 to 9 into a 3 x 2 x 5 array
x = np.random.randint(10, size=(3, 2, 5))
print('x is: \n', x, '\n')
y = np.random.randint(10, size=(2, 5))
print('y is: \n', y, '\n')
z = x + y
print('z is: \n', z)
```

```
x is: 
 [[[4 3 0 5 7]
  [5 6 6 2 1]]

 [[3 2 8 5 6]
  [6 0 6 3 0]]

 [[1 8 5 8 2]
  [1 7 8 0 0]]] 

y is: 
 [[2 1 2 8 2]
 [1 6 6 1 8]] 

z is: 
 [[[ 6  4  2 13  9]
  [ 6 12 12  3  9]]

 [[ 5  3 10 13  8]
  [ 7  6 12  4  8]]

 [[ 3  9  7 16  4]
  [ 2 13 14  1  8]]]
```

2.3 Tensor dot

The dot operations, also called tensor product, is very similiar to vector/matrix multiplication.

In [18]:

```python
import numpy as np
x = np.random.randint(10, size=(2, 5))
print('x is: \n', x, '\n')
y = np.random.randint(10, size=(5, 3))
print('y is: \n', y, '\n')
z = np.dot(x, y)
print('z is: \n', z)
```

```
x is: 
 [[3 2 7 0 1]
 [3 7 5 8 8]] 

y is: 
 [[3 1 1]
 [6 2 3]
 [4 7 5]
 [2 4 5]
 [5 5 7]] 

z is: 
 [[ 54  61  51]
 [127 124 145]]
```

In [19]:

```python
#element wise multiply
x = np.random.randint(10, size=(2, 5))
print('x is: \n', x, '\n')
y = np.random.randint(10, size=(2, 5))
print('y is: \n', y, '\n')
z = x*y
print('z is: \n', z)
```

```
x is: 
 [[4 4 2 9 4]
 [7 2 1 6 4]] 

y is: 
 [[9 6 0 9 1]
 [0 9 5 5 4]] 

z is: 
 [[36 24  0 81  4]
 [ 0 18  5 30 16]]
```

2.4 Tensor reshaping

Rearranging the rows and columns of a tensor to match a target shape.

In [20]:
```
x = np.array([[0., 1.],
              [2., 3.],
              [4., 5.]])
print(x, '\n')
print('The shape of x is:', x.shape)
```

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

The shape of x is: (3, 2)
```

In [21]:
```
x.dtype
```

Out[21]:
```
dtype('float64')
```

In [22]:
```
x = x.reshape((6, 1))
print(x, '\n')
print('The shape of x is:', x.shape)
```

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

The shape of x is: (6, 1)
```

A special case of reshaping that's commonly encountered is transposition.

In [23]:
```python
y = np.zeros((3, 2))
print('y is: \n', y, '\n')
print('The shape of y is:', y.shape,'\n')
y_t = np.transpose(y)
print('y_t is: \n', y_t, '\n')
print('The shape of y_t is:', y_t.shape,'\n')
```

```
y is:
 [[0. 0.]
 [0. 0.]
 [0. 0.]]

The shape of y is: (3, 2)

y_t is:
 [[0. 0. 0.]
 [0. 0. 0.]]

The shape of y_t is: (2, 3)
```

3. Exercise

1. Provide an example of 0D tensor, 1D tensor, 2D tensor and 3D tensor respectively.

In [24]:
```python
#Task 1: 0D tensor
x = np.array(12)
print(x)
```

```
12
```

In [25]:
```python
#Task 2: 1D tensor
x = np.array([3,4,7,8])
print(x)
```

```
[3 4 7 8]
```

In [26]:

```
#Task 3: 2D tensor
x = np.array([[5, 78, 2, 34],
             [6, 79, 3, 35],
             [7, 80, 4, 36]])
print(x)
```

```
[[ 5 78  2 34]
 [ 6 79  3 35]
 [ 7 80  4 36]]
```

In [27]:

```
#Task 4: 3D tensor
x = np.array([[[5, 78, 2],
               [6, 79, 3],
               [7, 80, 4]],
              [[5, 78, 2],
               [6, 79, 3],
               [7, 80, 4]],
              [[5, 78, 2],
               [6, 79, 3],
               [7, 80, 4]]])

print(x)
```

```
[[[ 5 78  2]
  [ 6 79  3]
  [ 7 80  4]]

 [[ 5 78  2]
  [ 6 79  3]
  [ 7 80  4]]

 [[ 5 78  2]
  [ 6 79  3]
  [ 7 80  4]]]
```

2. Provide a Tensor Slicing Example.

In [28]:
```
# Task 1: Tensor Slicing
y = x[0:2]
print(y)
```

```
[[[ 5 78  2]
  [ 6 79  3]
  [ 7 80  4]]

 [[ 5 78  2]
  [ 6 79  3]
  [ 7 80  4]]]
```

3. Provide an example for each tensor operation learnt in this Practical.

In [29]:
```
# Task 1: Element Wise Operation
x1 = np.array([3,4,7,8])
x2 = np.array([3,4,7,8])
x3 = x1 + x2

print(x3)
```

```
[ 6  8 14 16]
```

In [30]:
```
# Task 2: Broadcasting
y1 = y+2
print(y1)
```

```
[[[ 7 80  4]
  [ 8 81  5]
  [ 9 82  6]]

 [[ 7 80  4]
  [ 8 81  5]
  [ 9 82  6]]]
```

In [31]:

```python
# Task 3: Tensor Dot
x1 = np.array([[5, 78, 2, 34],
             [6, 79, 3, 35],
             [7, 80, 4, 36]])

x2 = np.array([1,2,3,4])

x3 = np.dot(x1, x2)
print(x3)
```

```
[303 313 323]
```

In [32]:

```python
# Task 4: Tensor Reshaping
print(x1.shape)
x2 = x1.reshape((6, 2))
print(x2)
print(x2.shape)
```

```
(3, 4)
[[ 5 78]
 [ 2 34]
 [ 6 79]
 [ 3 35]
 [ 7 80]
 [ 4 36]]
(6, 2)
```