Numpy Array và Pytorch/Tensorflow Tensor

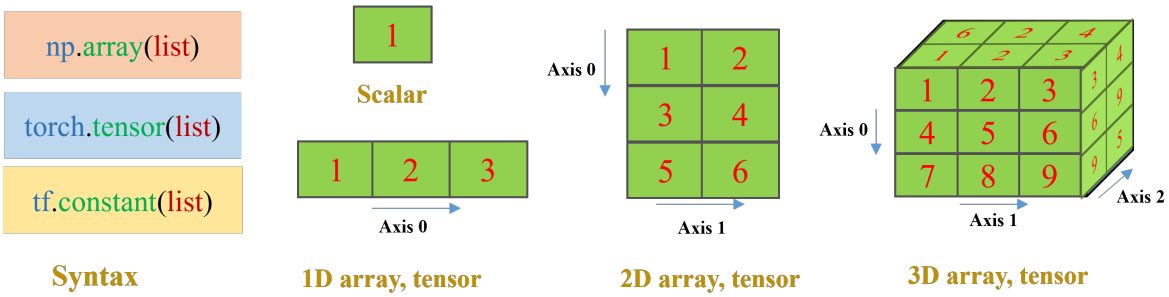
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Array trong Numpy và Tensor trong các thư viện Pytorch, Tensorflow là những thành phần nền tảng cốt lõi cho việc tính toán của các thư viện này. Trong đó:

- **Array** là một cấu trúc dữ liệu đa chiều mạnh mẽ, giúp lưu trữ và thực hiện các phép toán toán học trên dữ liệu số. Array có thể là mảng 0 chiều (scalar), mảng một chiều (vector), mảng hai chiều (ma trận), hoặc nhiều chiều hơn.
- **Tensor trong Pytorch và Tensorflow** là một cấu trúc dữ liệu nhiều chiều, tương tự như array trong Numpy, nhưng được thiết kế để tương thích với học sâu và tính toán song song, đặc biệt là trên các thiết bị như GPU và TPU. Trong Pytorch và Tensorflow, tensors là đơn vị cơ bản để lưu trữ và xử lý dữ liệu.

Có nhiều cách để tạo tensor (hay array trong Nympy), ta có thể sử dụng cú pháp sau đây để khởi tạo chúng từ list Python.



Hình 1: Minh họa và cú pháp tạo Array và Tensor.

Khi làm việc với cấu trúc dữ liệu tensor, ta có thể kiểm tra thuộc tính của chúng thông qua một số phương thức sau:

- **shape** cho biết kích thước của mảng hoặc tensor, tức là số phần tử trong mỗi chiều của chúng.
- **dtype** cho biết kiểu dữ liệu của các phần tử trong mảng hoặc tensor.
- **type** cho biết kiểu của đối tượng mảng hoặc tensor.
- **device** chỉ có ở trong Pytorch và Tensorflow, và nó cho biết nơi lưu trữ tensor, có thể là CPU hoặc GPU.

2. Bài tập

Câu 1: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor từ danh sách 1 chiều?

Câu 2: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor từ danh sách 2 chiều. Sau đó thực hiện kiểm tra thuộc tính shape, dtype, type, device từ các array, tensor vừa tạo ?

3. Đáp án

Có nhiều cách khác nhau để ta tạo array hay tensor. Cách đầu tiên, ta tạo chúng từ list-kiểu dữ liệu quen thuộc trong Python.

```python
import numpy as np
import torch
import tensorflow as tf

# Tạo một danh sách 1 chiều
list_1D = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Tạo một mảng 1 chiều từ danh sách
arr_1D = np.array(list_1D)
print("Mảng 1 chiều NumPy:\n",
      arr_1D, type(arr_1D))

# Tạo một tensor PyTorch từ danh sách
tensor_1D_pt = torch.tensor(list_1D)
print("Tensor PyTorch 1 chiều:\n",
      tensor_1D_pt)

# Tạo một tensor TensorFlow từ danh sách
tensor_1D_tf = tf.convert_to_tensor(
                                    list_1D)
print("Tensor TensorFlow 1 chiều:\n",
      tensor_1D_tf)
```

```
================ Output ===============
Mảng 1 chiều NumPy:
 [1 2 3 4 5 6 7 8 9]
 <class 'numpy.ndarray'>

Tensor PyTorch 1 chiều:
 tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])

Tensor TensorFlow 1 chiều:
 tf.Tensor([1 2 3 4 5 6 7 8 9],
           shape=(9,), dtype=int32)
=======================================
```

Ví dụ trên thực hiện tạo array, tensor từ một list Python. Để tạo array chúng ta dùng cú pháp **np.array(list_1D)**, với pytorch thì ta dùng **torch.tensor(list_1D)** và với Tensorflow ta dùng **tf.convert_to_tensor(list_1D)** hoặc **tf.constant(list_1D)**.

Đối với việc tạo array, tensor từ list 2D, chúng ta cũng thực hiện tương tự cách trên, Ví dụ Numpy:

```python
# NumPy code
import numpy as np
# Tạo một danh sách 2 chiều
list_2D = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]
# Tạo một mảng 2 chiều
arr_2D = np.array(list_2D)
# Kiểm tra hình dạng, kiểu dữ liệu và loại
print("Hình dạng của mảng: ",
      arr_2D.shape)
print("Kiểu dữ liệu của mảng: ",
      arr_2D.dtype)
print("Loại của mảng: ",
      type(arr_2D))
print("Mảng:\n", arr_2D)
```

```
================ Output ===============
Hình dạng của mảng:  (3, 3)

Kiểu dữ liệu của mảng:  int32

Loại của mảng:  <class 'numpy.ndarray'>

Mảng:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
=======================================
```

Trong đoạn mã Numpy trên, chúng ta bắt đầu bằng việc tạo ra một danh sách 2D **(list_2D)** đại diện cho một ma trận có 3 hàng và 3 cột. Sau đó, chúng ta sử dụng Numpy để chuyển danh sách này thành một mảng 2 chiều **(arr_2D)**. Dòng mã tiếp theo sử dụng các hàm tích hợp trong Numpy để in ra hình dạng (**shape**), kiểu dữ liệu (**dtype**), và kiểu của mảng (**type**). Cuối cùng, chúng ta in ra nội dung của mảng để kiểm tra kết quả

Ví dụ Pytorch:

```python
# PyTorch code
import torch

# Tạo một danh sách 2 chiều
list_2D = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]
# Tạo một tensor 2 chiều
tensor_2D_pt = torch.tensor(list_2D)

# Kiểm tra hình dạng, kiểu dữ liệu, loại,
    thiết bị lưu trữ của tensor
print("Hình dạng của tensor: ",
      tensor_2D_pt.shape)
print("Kiểu dữ liệu của tensor: ",
      tensor_2D_pt.dtype)
print("Loại của tensor: ",
      type(tensor_2D_pt))
print("Thiết bị lưu trữ của tensor: ",
      tensor_2D_pt.device)
print("Tensor:\n", tensor_2D_pt)
```

```
================ Output ===============
Hình dạng của tensor:  torch.Size([3, 3])

Kiểu dữ liệu của tensor:  torch.int64

Loại của tensor:  <class 'torch.Tensor'>

Thiết bị lưu trữ của tensor:  cpu

Tensor:
 tensor([[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]])
=======================================
```

Trong đoạn mã Pytorch trên, chúng ta bắt đầu bằng cách tạo một danh sách 2D (**list_2D**) đại diện cho ma trận 3x3. Sau đó, chúng ta sử dụng Pytorch để chuyển đổi danh sách này thành tensor 2 chiều (**tensor_2D_pt**).
Dòng mã tiếp theo sử dụng các thuộc tính tích hợp trong Pytorch để in ra hình dạng (**shape**), kiểu dữ liệu (**dtype**), kiểu của tensor (**type**), và thiết bị lưu trữ của tensor (**device**). Cuối cùng, chúng ta in ra nội dung của tensor để kiểm tra kết quả.
Ví dụ Tensorflow:

```python
import tensorflow as tf

# Tạo một danh sách 2 chiều
list_2D = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]
# Tạo một tensor 2 chiều từ danh sách
tensor_2D_tf = tf.convert_to_tensor(
                                list_2D)

# Kiểm tra hình dạng, kiểu dữ liệu, loại,
# và thiết bị mà tensor được lưu trữ
print("Hình dạng của tensor: ",
      tensor_2D_tf.shape)
print("Kiểu dữ liệu của tensor: ",
      tensor_2D_tf.dtype)
print("Loại của tensor: ",
      type(tensor_2D_tf))
print("Thiết bị mà tensor được lưu trữ: ",
      tensor_2D_tf.device)
print(tensor_2D_tf)
```

```
================ Output ===============
Hình dạng của tensor:  (3, 3)

Kiểu dữ liệu của tensor:  <dtype: 'int32'>

Loại của tensor:  <class 'tensorflow.
    python.framework.ops.EagerTensor'>

Thiết bị mà tensor được lưu trữ:  /job:
    localhost/replica:0/task:0/device:CPU
    :0

tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
=======================================
```

Tương tự như Pytorch, trong đoạn mã Tensorflow trên, chúng ta bắt đầu bằng cách tạo một danh sách

2D (**list_2D**) đại diện cho ma trận 3x3. Sau đó, chúng ta sử dụng Tensorflow để chuyển đổi danh sách này thành một tensor 2 chiều (**tensor_2D_tf**) bằng hàm **tf.convert_to_tensor()**.
Dòng mã tiếp theo sử dụng các thuộc tính tích hợp trong Tensorflow để in ra hình dạng (**shape**), kiểu dữ liệu (**dtype**), kiểu của tensor (**type**), và thiết bị lưu trữ của tensor (**device**). Cuối cùng, chúng ta in ra nội dung của tensor để kiểm tra kết quả.

Các Hàm Khởi Tạo Numpy Array và Pytorch/Tensorflow Tensor - Phần 1

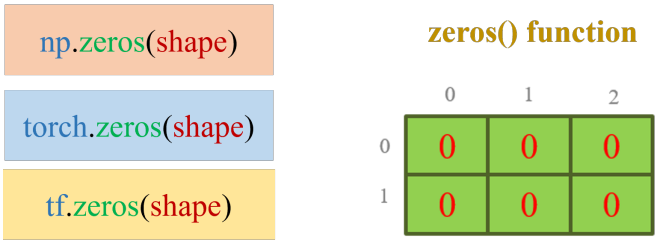
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Khi lập trình với các thư viện Numpy, Pytorch, Tensorflow có một số cách để nhanh chóng tạo ra các array với giá trị, kích thước khác nhau. Trong bài tập này, chúng ta sẽ tìm hiểu các sử dụng 3 hàm **zeros**, **ones**, **full**.

a) ***zeros***

zeros là hàm thực hiện chức năng tạo array, tensor toàn giá trị 0 với đầu vào là kích thước và kiểu dữ liệu ta muốn. Cả 3 thư viện đều sử dụng hàm trên, với cú pháp tương tự nhau.

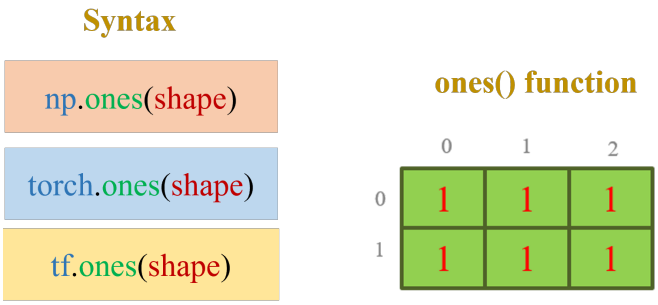


Hình 1: Minh họa và cú pháp sử dụng hàm zeros.

Nhìn chung, hàm **zeros** chủ yếu dùng để khởi tạo mảng hoặc tensor với các giá trị 0, thường được dùng trong quá trình xây dựng các mô hình máy học và thực hiện các phép toán số học.

b) ***ones***

Tương tự với **zeros**, hàm **ones** tạo mảng chứa toàn số 1 với đầu vào là kích thước do người dùng chỉ định.

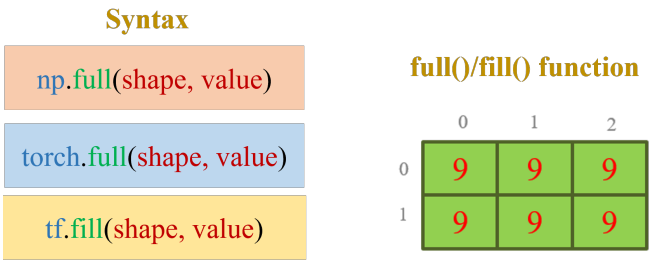


Hình 2: Minh họa và cú pháp sử dụng hàm ones.

c) ***full, fill***

Hàm **full** giúp chúng ta tạo array, tensor(Pytorch) với phần tử là giá trị chỉ định, đầu vào của hàm **full** gồm kích thước array và giá trị hằng số muốn tạo. Khác với hai thư viện trên, Tensorflow sử dụng hàm **fill**.

Hàm **full** với Numpy, Pytorch hay **fill** với Tensorflow giúp tạo array hoặc tensor với kích thước và giá trị được chỉ định, có ích khi ta cần khởi tạo các cấu trúc dữ liệu với giá trị đồng nhất.



Hình 3: Minh họa và cú pháp sử dụng hàm full/fill.

2. Bài tập

Câu 1: Hãy viết chương trình sử dụng hàm zeros tạo Numpy array, Tensorflow tensor, Pytorch tensor chỉ chứa giá trị là số 0 với kích thước (3, 4)?

Câu 2: Hãy viết chương trình sử dụng hàm ones tạo Numpy array, Tensorflow tensor, Pytorch tensor chỉ chứa giá trị là số 1 với kích thước (3, 4)?

Câu 3: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor chỉ chứa giá trị là số 5 với kích thước (3, 4)?

3. Đáp án

Câu 1: ***zeros***

Chương trình sau tạo array với kích thước (3, 4), đối với thư viện Numpy sử dụng cú pháp **np.zeros**, bên trong hàm này chúng ta truyền vào kích thước array chúng ta mong muốn là (3, 4).

```
# Numpy code
import numpy as np

# Tạo array toàn số 0
arr_zeros = np.zeros((3, 4))
print(arr_zeros)
```

```
================ Output ===============
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

=======================================
```

Tương tự như thư viện Numpy, Pytorch sử dụng cú pháp ***torch.zeros***

```
#Pytorch code
import torch

# Tạo tensor toàn số 0
tensor_zeros = torch.zeros((3, 4))
print(tensor_zeros)
```

```
================ Output ===============
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])

=======================================
```

Tương tự, Tensorflow sử dụng cú pháp **tf.zeros**

```
# Tensorflow code
import tensorflow as tf

# Tạo tensor toàn số 0
tensor_zeros = tf.zeros((3, 4))
print(tensor_zeros)
```

```
================ Output ===============
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]], shape=(3, 4), dtype=
   float32)
=======================================
```

Câu 2: ***ones***

Chương trình sau tạo array với kích thước (3, 4), đối với thư viện Numpy sử dụng cú pháp ***np.ones***

```python
# Numpy code
import numpy as np

# Tạo array toàn số 1
arr_ones = np.ones((3, 4))
print(arr_ones)
```

```
================ Output ===============
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

=======================================
```

Với Pytorch và Tensorflow cú pháp thực hiện tương tự với cú pháp torch.ones, tf.ones

```python
#PyTorch code
import torch

# Tạo tensor toàn số 1
tensor_ones = torch.ones((3, 4))
print(tensor_ones)
```

```
================ Output ===============
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])

=======================================
```

```python
#TensorFlow code
import tensorflow as tf


# Tạo tensor toàn số 1
tensor_ones = tf.ones((3, 4))
print(tensor_ones)
```

```
================ Output ===============
tf.Tensor(
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]], shape=(3, 4), dtype=
   float32)
=======================================
```

Câu 3: ***full, fill***

Chương trình sau tạo array với kích thước (3, 4), giá trị hằng số là 5. Đối với thư viện Numpy sử dụng cú pháp ***np.full***

```python
# Numpy code
import numpy as np

# Tạo array toàn số 5
arr_full = np.full((3, 4), 5)
print(arr_full)
```

```
================ Output ===============
[[5 5 5 5]
 [5 5 5 5]
 [5 5 5 5]]

=======================================
```

Tương tự như thư viện Numpy, Pytorch sử dụng cú pháp ***torch.full***

```python
#Pytorch code
import torch

# Tạo tensor toàn số 5
tensor_full = torch.full((3, 4), 5)
print(tensor_full)
```

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

=======================================
```

Khác với hai thư viện trên, Tensorflow sử dụng hàm **fill** để tạo một tensor với giá trị được chỉ định.

```python
# Tensorflow code
import tensorflow as tf

# Tạo tensor toàn số 5
tensor_full = tf.fill((3, 4), 5)
print(tensor_full)
```

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

Các Hàm Khởi Tạo Numpy Array và Pytorch/Tensorflow Tensor - Phần 2

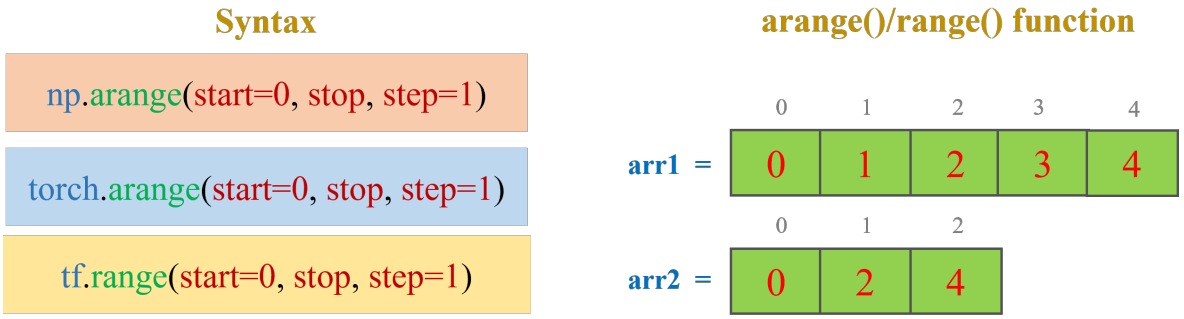
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Trong bài tập này, chúng ta tiếp tục tìm hiểu các hàm có chức năng tạo array, tensor đặc biệt. Chúng ta sẽ tìm hiểu và sử dụng các hàm **arange**, **range**, **eye**, và **random**.

a) ***arange, range***

Trong Numpy, Pytorch, hàm **arange** được sử dụng để tạo một mảng chứa dãy số có giá trị tăng dần với khoảng cách cố định. Trong Tensorflow chúng ta sẽ sử dụng hàm **range**. Cú pháp như sau:

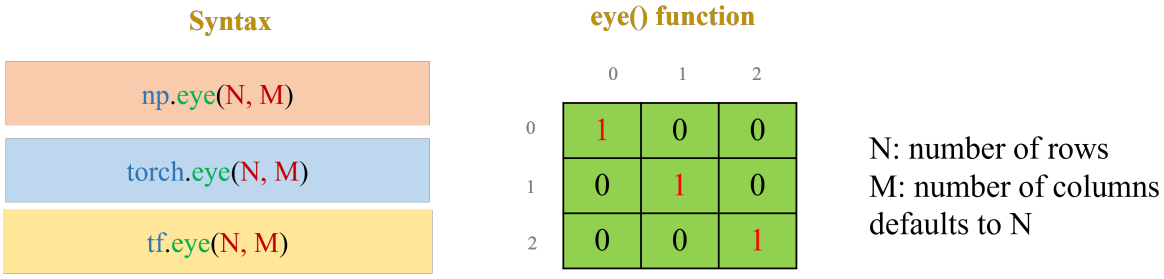


Hình 1: Minh họa và cú pháp sử dụng hàm arange, range.

Hàm **arange** (hoặc **range** trong Tensorflow) rất hữu ích khi ta muốn tạo ra một dãy số tăng dần với các giá trị cách đều nhau.

b) ***eye***

Hàm **eye** được sử dụng để tạo ma trận đơn vị, tức là một ma trận vuông có tất cả các phần tử trên đường chéo chính có giá trị 1, các phần tử còn lại là 0. Cú pháp sử dụng như sau:

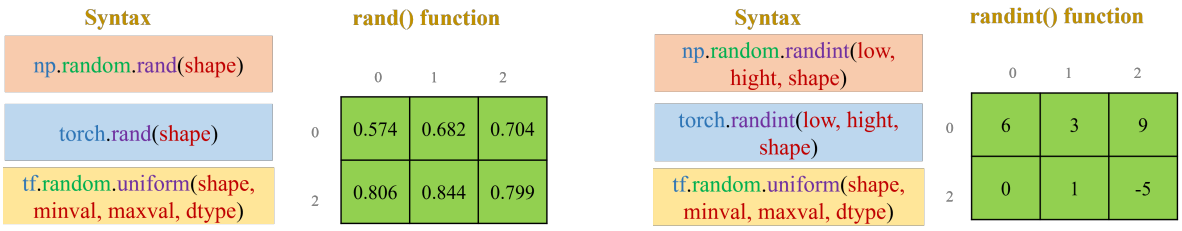


Hình 2: Minh họa và cú pháp sử dụng hàm eye.

Hàm **eye** giúp tạo array hoặc tensor đơn vị, đặc biệt hữu ích trong nhiều ứng dụng toán học và xử lý dữ liệu.

c) ***random***

Trong thư viện Numpy, Pytorch cung cấp hàm **rand** để tạo mảng với các giá trị ngẫu nhiên trong khoảng [0.0, 1.0) với kích cỡ (shape) được chỉ định. Ngoài ra có thể sử dụng hàm **randint** để tạo mảng với các số nguyên ngẫu nhiên. Đối với thư viện Tensorflow chúng ta sử dụng hàm **random.uniform**.



Hình 3: Minh họa và cú pháp sử dụng hàm random.

Các hàm này giúp việc tạo dữ liệu ngẫu nhiên trở nên dễ dàng trong quá trình phát triển mô hình Machine Learning, Deep Learning.

2. Bài tập

Câu 1: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor trong khoảng [0, 10) với step=1?

Câu 2: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor là ma trận đơn vị với kích thước (3, 3).

Câu 3: Hãy viết chương trình tạo Numpy array, Tensorflow tensor, Pytorch tensor ngẫu nhiên trong khoảng giá trị [0, 1] với kích thước (3, 4). Tiếp theo hãy tạo array, tensor với các giá trị là số nguyên trong khoảng [-10, 10]. Lưu ý trong bài tập này, các bạn sẽ sử dụng seed = 2024 để dảm bảo ra kết quả giống đáp án, cách sử dụng như sau:

```
# Numpy code
import numpy as np
np.random.seed(2024)

# Pytorch code
import torch
torch.manual_seed(2024)

# Tensorflow code
import tensorflow as tf
tf.random.set_seed(2024)
```

3. Đáp án

Câu 1: ***arange, range***

Trong numpy chúng ta sử dụng hàm **np.arange**

```
# Numpy code
import numpy as np

# Tạo array trong khoảng [0, 10),
# bước nhảy 1
arr_arange = np.arange(10)
print(arr_arange)
```

```
================ Output ================
[0 1 2 3 4 5 6 7 8 9]




========================================
```

Trong PyTorh, để tạo tensor chứa dãy số, ta sử dụng hàm **torch.arange**:

```
#Pytorch code
import torch

# Tạo tensor trong khoảng [0, 10),
# bước nhảy 1
tensor_arange = torch.arange(10)
print(tensor_arange)
```

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



=======================================
```

Trong Tensorflow, ta có thể sử dụng hàm **tf.range** để thực hiện chức năng tương tự:

```
# Tensorflow code
import tensorflow as tf

# Tạo tensor trong khoảng [0, 10),
# bước nhảy 1
tensor_arange = tf.range(10)
print(tensor_arange)
```

```
================ Output ===============
tf.Tensor([0 1 2 3 4 5 6 7 8 9],
shape=(10,), dtype=int32)



=======================================
```

Câu 2: ***eye***

Trong numpy chúng ta sử dụng hàm np.eye để tạo ma trận đơn vị, trong đó giá trị chúng ta truyền vào là số lượng hàng của ma trận đầu ra, số lượng cột mặc định bằng số lượng hàng, nếu muốn số lượng cột khác bạn cần truyền vào hàm eye số lượng cột chỉ định.

```
# Numpy code
import numpy as np

# Tạo array với đường chéo chính là 1,
# còn lại là 0
arr_eye = np.eye(3)
print(arr_eye)
```

```
================ Output ===============
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]


=======================================
```

Trong Pytorch, để tạo tensor đơn vị, ta sử dụng hàm **torch.eye**:

```
#Pytorch code
import torch

# Tạo tensor với đường chéo chính là 1,
# còn lại là 0
tensor_eye = torch.eye(3)
print(tensor_eye)
```

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


=======================================
```

Trong Tensorflow, hàm **eye** cũng được sử dụng để tạo constant tensor đơn vị:

```
# Tensorflow code
import tensorflow as tf

# Tạo tensor với đường chéo chính là 1,
# còn lại là 0
tensor_eye = tf.eye(3)
print(tensor_eye)
```

```
================ Output ===============
tf.Tensor(
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]], shape=(3, 3), dtype=float32)

=======================================
```

Câu 3: ***random***

Trong Numpy, chúng ta sẽ sử dụng hàm **random.rand** để tạo mảng với các giá trị ngẫu nhiên trong khoảng giá trị mặc định là [0, 1). Để tạo mảng với giá trị ngẫu nhiên là số nguyên chúng ta dùng **random.randint**, khi sử dụng hàm này ta cần truyền vào khoảng giá trị lấy ngẫu nhiên và kích thước đầu ra mong muốn.

```python
#Numpy code
import numpy as np

# Đặt seed để đảm bảo kết quả ngẫu nhiên
# có thể tái tạo được
np.random.seed(2024)

# Tạo một mảng với các giá trị ngẫu nhiên trong khoảng [0, 1) với kích thước (3, 4)
arr_rand = np.random.rand(3, 4)
print("Mảng ngẫu nhiên trong khoảng [0, 1):\n", arr_rand)

# Tạo một mảng với các giá trị ngẫu nhiên trong khoảng [-10, 10)
arr_randint = np.random.randint(-10, 10, size=(3, 4))
print("Mảng ngẫu nhiên trong khoảng [-10, 10):\n", arr_randint)
```

```
================ Output ===============
Mảng ngẫu nhiên trong khoảng [0, 1):
 [[0.58801452 0.69910875 0.18815196
   0.04380856]
 [0.20501895 0.10606287 0.72724014
   0.67940052]
 [0.4738457  0.44829582 0.01910695
   0.75259834]]

Mảng ngẫu nhiên trong khoảng [-10, 10):
 [[ 5  1 -3  8]
 [-1 -4  0 -9]
 [-5  9 -6 -2]]
=======================================
```

Trong Pytorch, các hàm tương tự cũng có sẵn thông qua module torch. **torch.rand** tạo tensor với các giá trị ngẫu nhiên từ phân phối đều trong khoảng [0.0, 1.0), **torch.randint** dùng để tạo tensor với các số nguyên ngẫu nhiên trong một khoảng cụ thể.

```python
# Pytorch code
import torch

# Thiết lập seed để đảm bảo kết quả ngẫu nhiên có thể tái tạo được
torch.manual_seed(2024)

# Tạo tensor với các giá trị ngẫu nhiên trong khoảng [0, 1) với kích thước (3, 4)
tensor_rand = torch.rand((3, 4))
print("Tensor ngẫu nhiên trong khoảng [0, 1):\n", tensor_rand)

# Tạo tensor với các giá trị ngẫu nhiên trong khoảng [-10, 10)
tensor_randint = torch.randint(-10, 10, size=(3, 4))
print("Tensor ngẫu nhiên trong khoảng [-10, 10):\n", tensor_randint)
```

```
================ Output ===============
Tensor ngẫu nhiên trong khoảng [0, 1):
 tensor([[0.5317, 0.8313, 0.9718, 0.1193],
        [0.1669, 0.3495, 0.2150, 0.6201],
        [0.4849, 0.7492, 0.1521, 0.5625]])

Tensor ngẫu nhiên trong khoảng [-10, 10):
 tensor([[  1,   8,   0,  -2],
        [ -9, -10,  -9,   0],
        [ -3,  -7,  -4,   8]])
=======================================
```

Trong Tensorflow, ta cũng có các hàm tương tự là **tf.random.uniform** để tạo tensor với các giá trị ngẫu nhiên trong khoảng [0.0, 1.0), và sử dụng **tf.random.uniform** với **dtype = tf.dtypes.int32** để tạo tensor với các số nguyên ngẫu nhiên trong một khoảng cụ thể.

```tensorflow
#TensorFlow code
import tensorflow as tf


# Thiết lập seed để đảm bảo kết quả ngẫu nhiên có thể tái tạo được
tf.random.set_seed(2024)

# Tạo tensor với các giá trị ngẫu nhiên trong khoảng [0, 1)
tensor_rand = tf.random.uniform((3, 4))
print("Tensor ngẫu nhiên trong khoảng [0, 1):\n", tensor_rand)

# Tạo tensor với các giá trị ngẫu nhiên trong khoảng [-10, 10)
tensor_randint = tf.random.uniform((3, 4), minval=-10, maxval=10, dtype=tf.dtypes.int32)
print("Tensor ngẫu nhiên trong khoảng [-10, 10):\n", tensor_randint)
```

```
================= Output ===============
Tensor ngẫu nhiên trong khoảng [0, 1):

 tf.Tensor(
[[0.90034294 0.19453335 0.36069036
   0.66361904]
 [0.76605344 0.2159369  0.6261736
   0.07380784]
 [0.22062695 0.934368   0.93327904
   0.69267046]],
 shape=(3, 4), dtype=float32)

Tensor ngẫu nhiên trong khoảng [-10, 10):
 tf.Tensor(
[[-3 -7  9  3]
 [ 2 -5 -3 -5]
 [ 4 -3 -3  5]],
 shape=(3, 4), dtype=int32)


=========================================
```

Numpy, Pytorch và Tensorflow
Hàm Stack và Concatenate

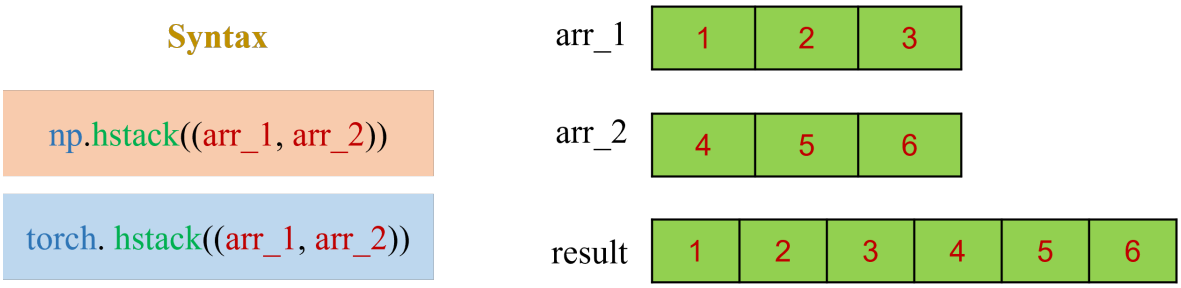
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Trong các thư viện Numpy, Pytorch, Tensorflow các hàm **hstack**, **vstack**, và **concatenate** được sử dụng để nối các array hoặc các tensor lại với nhau theo các hướng khác nhau. Trong bài tập này, chúng ta sẽ tìm hiểu cách sử dụng ba hàm trên.

a) ***hstack***

Hàm **hstack** trong NumPy, PyTorch được sử dụng để nối các array hoặc tensor theo chiều ngang, tức là nối chúng thành một cấu trúc dữ liệu lớn hơn theo chiều thứ hai. Quá trình này giúp kết hợp dữ liệu từ các nguồn khác nhau hoặc mở rộng kích thước của cấu trúc dữ liệu hiện có.

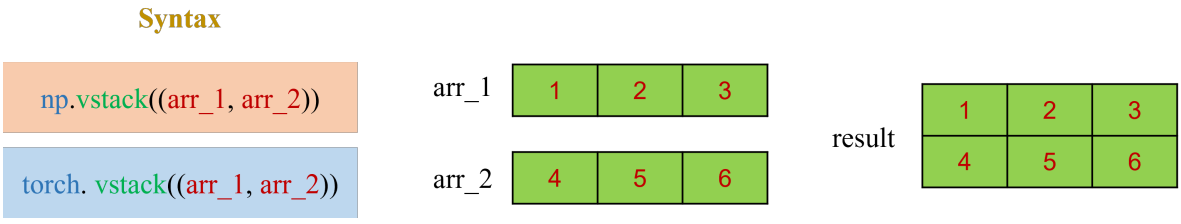


Hình 1: Minh họa và cú pháp sử dụng hstack

Lưu ý khi sử dụng **hstack** là các array hoặc tensor cần phải có cùng độ dài trong chiều dọc (cùng số hàng). Nếu các array/tensor không có cùng độ dài trong chiều dọc sẽ gây ra lỗi khi thực thi chương trình.

b) ***vstack***

Hàm **vstack** trong NumPy, PyTorch được sử dụng để nối các array hoặc tensor theo chiều dọc, tức là nối chúng thành một cấu trúc dữ liệu lớn hơn theo chiều thứ nhất. Quá trình này giúp kết hợp dữ liệu từ các nguồn khác nhau hoặc mở rộng kích thước của cấu trúc dữ liệu hiện có.

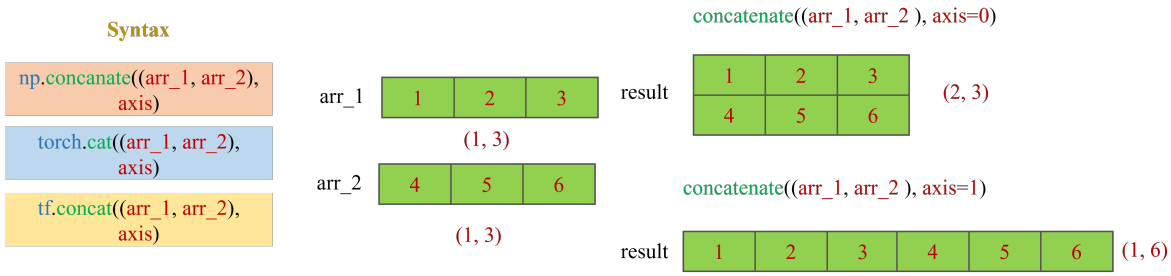


Hình 2: Minh họa và cú pháp sử dụng vstack

Lưu ý khi sử dụng **vstack** là các array hoặc tensor cần phải có cùng số cột. Nếu các array/tensor không có cùng số cột sẽ gây ra lỗi khi thực thi chương trình.

c) ***concanate***

Trong Tensorflow không có hàm **hstack** và **vstack**, tuy nhiên chúng ta có thể sử dụng hàm **concanate** để thay thế. Khác với hai hàm trên, hàm **concatenate** được sử dụng để nối các array hoặc tensor theo một chiều cụ thể.



Hình 3: Minh họa và cú pháp sử dụng concanate

Khi sử dụng concanate các bạn cần lưu ý, đối với việc nối hai array, tensor 1D theo chiều 1 sẽ gặp lỗi, lí do là vì các array, tensor 1D chỉ có 1 chiều là chiều 0. Chính vì vậy mầ ta cần chuyển đổi chúng sang dạng 2D trước khi nối chúng lại theo chiều 1. Một lựa chọn dễ dàng hơn là sử dụng vstack.

2. Bài tập

Câu 1: Hãy viết chương trình tạo Numpy 2 array, Pytorch tensor với từ list 1 có giá trị [1, 2, 3], list 2 có giá trị [4, 5, 6]. Sau đó sử dụng hàm **hstack** để nối theo trục ngang và **vstack** để nối theo trục dọc?

Câu 2: Hãy viết chương trình tạo Numpy 2 array, Pytorch tensor, Tensorflow tensor từ list 2D là [[1 , 2 , 3] , [4 , 5 , 6]] và [[7 , 8 , 9] , [10 , 11 , 12]]. Sau đó sử dụng hàm **concanate** để nối theo trục ngang và vstack để nối theo trục dọc?

3. Đáp án

Câu 1: ***hstack***

Chương trình sử dụng hstack được sử dụng để nối các array theo chiều ngang trong Numpy:

```python
#Numpy code
import numpy as np

# Tạo hai mảng
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])

# Nối hai mảng theo trục ngang (axis=0)
arr_3 = np.hstack((arr_1, arr_2))

# In kết quả
print("Mảng 1:\n", arr_1)
print("Mảng 2:\n", arr_2)
print("Mảng sau khi nối theo trục ngang:\n
    ", arr_3)
```

```
================ Output ===============
[1 2 3 4 5 6]
=======================================
```

Trong Pytorch sử dụng tương tự như Numpy với cú pháp **torch.hstack**.

```python
#Pytorch code
import torch

# Tạo hai tensor
tensor_1 = torch.tensor([1, 2, 3])
tensor_2 = torch.tensor([4, 5, 6])

# Nối hai tensor theo trục ngang
tensor_3 = torch.hstack((tensor_1, tensor_2))
print("Tensor 1:\n", tensor_1)
print("Tensor 2:\n", tensor_2)
print("Tensor sau khi nối theo trục ngang:\n", tensor_3)
```

```
================ Output ===============
Tensor 1:
 tensor([1, 2, 3])

Tensor 2:
 tensor([4, 5, 6])

Tensor sau khi nối theo trục ngang:
 tensor([1, 2, 3, 4, 5, 6])

=======================================
```

Câu 1: ***vstack***

Hàm **vstack** được sử dụng để nối (hoặc "stack") các array hoặc tensor theo chiều dọc (theo trục hàng). Trong Numpy, **vstack** được sử dụng để nối các array theo chiều dọc. Chương trình:

```python
# Numpy code
import numpy as np
# Tạo hai array
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])
# Nối hai array theo chiều dọc
arr_3 = np.vstack((arr_1, arr_2))
# In kết quả
print("Array 1:\n", arr_1)
print("Array 2:\n", arr_2)
print("Array sau khi nối theo chiều dọc:\n", arr_3)
```

```
================ Output ===============
Array 1:
 [1 2 3]

Array 2:
 [4 5 6]

Array sau khi nối theo chiều dọc:
 [[1 2 3]
 [4 5 6]]

=======================================
```

```python
#Pytorch code
import torch
# Tạo hai tensor
tensor_1 = torch.tensor([1, 2, 3])
tensor_2 = torch.tensor([4, 5, 6])
# Nối hai tensor theo chiều dọc bằng cách sử dụng torch.vstack
tensor_3 = torch.vstack((tensor_1, tensor_2))
# In kết quả
print("Tensor 1:\n", tensor_1)
print("Tensor 2:\n", tensor_2)
print("Tensor sau khi nối theo chiều dọc:\n", tensor_3)
```

```
================ Output ===============
Tensor 1:
 tensor([1, 2, 3])

Tensor 2:
 tensor([4, 5, 6])

Tensor sau khi nối theo chiều dọc:
 tensor([[1, 2, 3],
        [4, 5, 6]])

=======================================
```

Câu 2: ***Concanate***

Trong Numpy, **concatenate** được sử dụng để nối các array theo chiều ngang và dọc. Chương trình:

```python
# Numpy code
import numpy as np
# Tạo hai array 2D
arr_1 = np.array([[1, 2, 3],
                  [4, 5, 6]])
arr_2 = np.array([[7, 8, 9],
                  [10, 11, 12]])
# Nối hai array theo chiều dọc axis = 0
arr_3 = np.concatenate((arr_1, arr_2),
    axis=0)
# Nối hai array theo chiều ngang axis = 1
arr_4 = np.concatenate((arr_1, arr_2),
    axis=1)
# In kết quả
print("Array 1:\n", arr_1)
print("Array 2:\n", arr_2)
print("Nối theo chiều dọc (axis=0):\n",
    arr_3)
print("Nối theo chiều ngang (axis=1):\n",
    arr_4)
```

```
================ Output ===============
Array 1:
 [[1 2 3]
 [4 5 6]]

Array 2:
 [[ 7  8  9]
 [10 11 12]]

Nối theo chiều dọc (axis=0):
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Nối theo chiều ngang (axis=1):
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]

=======================================
```

Trong chương trình trên ta sử dụng **np.concatenate** để nối hai array theo chiều dọc (axis=0) và chiều ngang (axis=1). Trong đó **arr_1 và arr_2** là hai array 2D được tạo bằng NumPy. Tiếp theo, **arr_3** được tạo bằng cách nối **arr_1** và **arr_2** theo chiều dọc (axis=0), nghĩa là nối theo hàng. Cuối cùng **arr_4** được tạo bằng cách nối **arr_1** và **arr_2** theo chiều ngang (axis=1), nghĩa là nối theo cột.
Trong Pytorch, để thực hiện nối tensor theo một chiều cụ thể, ta sử dụng torch.cat với tham số dim, trong đó dim = 1 sẽ nối theo chiều ngang, dim = 0 sẽ nối theo chiều dọc:

```python
#Pytorch code
import torch
# Tạo hai tensor 2D
tensor_1 = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])
tensor_2 = torch.tensor([[7, 8, 9],
                         [10, 11, 12]])
# Nối hai tensor theo chiều dọc axis = 0
tensor_3 = torch.cat((tensor_1, tensor_2),
     dim=0)
# Nối hai tensor theo chiều ngang axis = 1
tensor_4 = torch.cat((tensor_1, tensor_2),
     dim=1)
# In kết quả
print("Tensor 1:\n", tensor_1)
print("Tensor 2:\n", tensor_2)
print("Nối theo chiều dọc (axis=0):\n",
    tensor_3)
print("Nối theo chiều ngang (axis=1):\n",
    tensor_4)
```

```
================ Output ===============
Tensor 1:
 tensor([[1, 2, 3],
        [4, 5, 6]])

Tensor 2:
 tensor([[ 7,  8,  9],
        [10, 11, 12]])

Nối theo chiều dọc (axis=0):
 tensor([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9],
        [10, 11, 12]])

Nối theo chiều ngang (axis=1):
 tensor([[ 1,  2,  3,  7,  8,  9],
        [ 4,  5,  6, 10, 11, 12]])

=======================================
```

Trong Tensorflow, **concatenate** tương đương với **tf.concat** với tham số axis để chỉ định kết nối theo chiều nào:

```python
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 2D
tensor_1 = tf.constant([[1, 2, 3],
                        [4, 5, 6]])
tensor_2 = tf.constant([[7, 8, 9],
                        [10, 11, 12]])

# Nối hai tensor theo chiều dọc axis = 0
tensor_3 = tf.concat((tensor_1, tensor_2),
    axis=0)

# Nối hai tensor theo chiều ngang axis = 1
tensor_4 = tf.concat((tensor_1, tensor_2),
    axis=1)

# In kết quả
print("Tensor 1:\n", tensor_1)
print("Tensor 2:\n", tensor_2)
print("Nối theo chiều dọc (axis=0):\n",
    tensor_3)
print("Nối theo chiều ngang (axis=1):\n",
    tensor_4)
```

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

Tensor 2:
 tf.Tensor(
[[ 7  8  9]
 [10 11 12]], shape=(2, 3), dtype=int32)

Nối theo chiều dọc (axis=0):
 tf.Tensor(
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]], shape=(4, 3), dtype=int32)

Nối theo chiều ngang (axis=1):
 tf.Tensor(
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]], shape=(2, 6),
 dtype=int32)
=======================================
```

Hàm **concatenate** là một công cụ linh hoạt, cho phép ta nối các array hoặc tensor theo bất kỳ trục nào mà ta mong muốn.

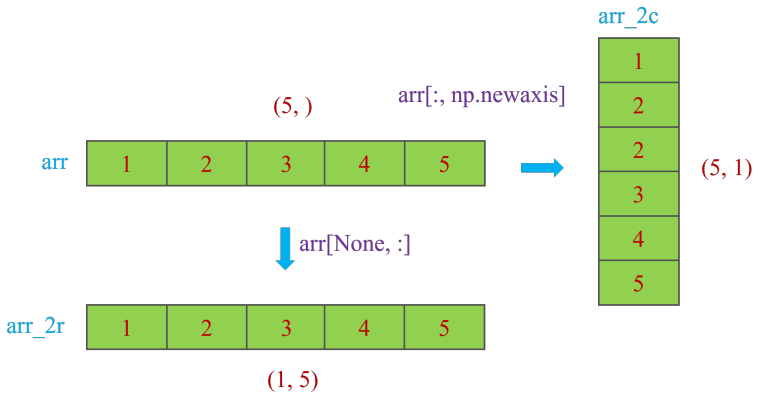
Các Hàm Quan Trọng Trong Numpy, Pytorch, Tensorflow - Phần 2

Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

a) **Thêm chiều dữ liệu**

Khi thực hiện tính toán với array, tensor, chúng ta sẽ thường xuyên phải điều chỉnh số chiều của chúng. Để thêm một chiều mới, chúng ta sử dụng **newaxis** hoặc **expand_dims**.



Hình 1: Minh họa cách thêm chiều dữ liệu

Ví dụ sau đây thực hiện thêm một chiều mới vào array sử dụng **newaxis**:

```python
#Numpy code
import numpy as np
# Tạo một array 1D
arr_1 = np.array([1, 2, 3])
# 1D -> 2D
arr_2 = arr_1[np.newaxis, :]
# 2D -> 5D
arr_3 = arr_2[np.newaxis, :, np.newaxis,
    :, np.newaxis]
print("array 1: ", arr_1, arr_1.shape)
print("array 2: ", arr_2, arr_2.shape)
print("array 3:\n ", arr_3, arr_3.shape)
```

```
================= Output ================
array 1:  [1 2 3] (3,)

array 2:  [[1 2 3]] (1, 3)

array 3:
  [[[[[1]
    [2]
    [3]]]]] (1, 1, 1, 3, 1)


=========================================
```

Trong ví dụ trên, ta tạo một array 1D có giá trị [1, 2, 3]. Tiếp theo ta sử dụng **np.newaxis** để thêm một chiều mới ở vị trí 0, chuyển từ array 1D thành array 2D. Cuối cùng ta dụng **np.newaxis** để thêm chiều mới, chuyển từ array 2D thành array 5D. Các chiều mới được thêm vào ở vị trí 0, 2, và 4.

Ở đây, dấu ":"Thể hiện chiều của array cũ, dấu hai chấm đầu tiên là chiều thứ nhất, dấu hai chấm thứ hai là chiều thứ 2 của array cũ. Ta có thể đặt dấu hai chấm này ở các vị trí khác nhau, tùy thuộc vào mục đích tạo array mới. "**np.newaxis**"là chiều mới ta muốn tạo, có thể đặt ở các vị trí khác nhau, ta cũng có thể thay thế nó bằng "**None**"hoặc "**...**".

Ngoài ra ta cũng có thể sử dụng cách thứ hai, **np.expand_dims** để thêm chiều mới cho array.

```python
# Numpy code
import numpy as np
# Tạo một array 1D
arr_1 = np.array([1, 2, 3])
# Thêm chiều mới, chuyển từ 1D -> 2D
arr_2 = np.expand_dims(arr_1, axis=0)
# Thêm nhiều chiều mới, chuyển từ 2D -> 5D
arr_3 = np.expand_dims(arr_2,
        axis=(0, 2, 4))
# In kết quả
print("array 1:", arr_1, arr_1.shape)
print("array 2:", arr_2, arr_2.shape)
print("array 3:\n", arr_3, arr_3.shape)
```

```
================ Output ===============
array 1: [1 2 3] (3,)

array 2: [[1 2 3]] (1, 3)

array 3:
 [[[[[1]
    [2]
    [3]]]]] (1, 1, 1, 3, 1)



=======================================
```

Trong Pytorch, Tensorflow, cách thêm chiều cũng thực hiện tương tự. Dưới đây là ví dụ Pytorch:

```python
#Pytorch code
import torch
# Tạo một tensor 1D
tensor_1 = torch.tensor([1, 2, 3])
# Thêm chiều mới, chuyển từ 1D -> 2D
tensor_2 = tensor_1[None, :]
# Thêm nhiều chiều mới, chuyển từ 2D -> 5D
tensor_3 = tensor_2[None, :, None, :, None
    ]
# In kết quả
print("tensor 1:", tensor_1, tensor_1.
    shape)
print("tensor 2:", tensor_2, tensor_2.
    shape)
print("tensor 3:\n", tensor_3, tensor_3.
    shape)
```

```
=======================================
tensor 1: tensor([1, 2, 3])
torch.Size([3])

tensor 2: tensor([[1, 2, 3]])
torch.Size([1, 3])

tensor 3:
 tensor([[[[[1],
           [2],
           [3]]]]])
           torch.Size([1, 1, 1, 3, 1])



=======================================
```

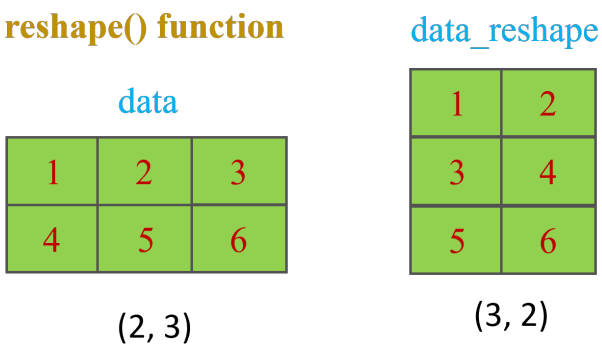
Ví dụ Tensorflow:

```python
#TensorFlow code
import tensorflow as tf
# Tạo một tensor 1D
tensor_1 = tf.constant([1, 2, 3])
# Thêm chiều mới, chuyển từ 1D -> 2D
tensor_2 = tf.expand_dims(tensor_1,
            axis=0)
# Thêm nhiều chiều mới, chuyển từ 2D -> 5D
tensor_3 = tensor_2[None, :, None, :, None
    ]
# In kết quả
print("tensor 1:", tensor_1, tensor_1.
    shape)
print("tensor 2:", tensor_2, tensor_2.
    shape)
print("tensor 3:\n", tensor_3, tensor_3.
    shape)
```

```
================ Output ===============
tensor 1: tf.Tensor([1 2 3], shape=(3,),
   dtype=int32) (3,)

tensor 2: tf.Tensor([[1 2 3]],
shape=(1, 3), dtype=int32) (1, 3)

tensor 3:
 tf.Tensor(
[[[[[1]
    [2]
    [3]]]]], shape=(1, 1, 1, 3, 1),
    dtype=int32) (1, 1, 1, 3, 1)



=======================================
```

b) ***reshape***

reshape là một hàm được sử dụng để thay đổi hình dạng của một array hoặc tensor mà không làm thay đổi dữ liệu bên trong nó. Hình dạng mới được chỉ định thông qua tham số của hàm **reshape**.



Hình 2: Minh họa cách sử dụng hàm reshape

Cú pháp sử dụng reshape:

```
# Numpy
np.reshape(input, newshape)

# Pytorch
torch.reshape(input, newshape)

# Tensorflow
torch.reshape(input, newshape)
```

Trong đó:

- input: là array, tensor đầu vào.
- newshape là hình dạng mới ta muốn chuyển đổi.

Trong ví dụ sau, ta sẽ sử dụng **reshape** để giảm chiều array, tensor:

```
# Numpy code
import numpy as np
# Tạo một array 2D
arr_2D = np.array([[1, 2, 3], [4, 5, 6]])
# Chuyển từ 2D -> 1D
arr_1D = np.reshape(arr_2D,
                    newshape=(6, ))
print("array 2D:\n", arr_2D, arr_2D.shape)
print("array 1D:\n", arr_1D, arr_1D.shape)
```

```
================ Output ===============
array 2D:
 [[1 2 3]
 [4 5 6]] (2, 3)

array 1D:
 [1 2 3 4 5 6] (6,)

=======================================
```

Trong ví dụ trên, ta tạo một array 2D có 2 hàng và 3 cột. Sử dụng **np.reshape** để thay đổi hình dạng của array từ 2D thành 1D với hình dạng mới là (6,), nghĩa là một array có 6 phần tử. Ở đây 6= 2x3, bằng với số phần tử của cũ, ta cần lưu ý khi thay đổi shape thì cần đảm bảo số lượng phần tử không được thay đổi khi **reshape**. Với Tensorflow, Pytorch cũng thực hiện tương tự.

2. Bài tập

Câu 1: Viết chương trình tạo một Numpy array, Pytorch tensor, Tensorflow tensor từ list 1D có giá trị [2, 3, 5, 7]. Sau đó thực hiện thêm một chiều mới ở vị trí 0 để tạo array, tensor 2D. Sau đó tiếp tục thêm các chiều mới ở vị trí 1, 3 và 4 để tạo array, tensor 5D.

Câu 2: Hãy viết chương trình để tạo một numpy array, một PyTorch tensor và một TensorFlow tensor từ một list 3D có giá trị như sau: [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]], [[13, 14, 15], [16, 17, 18]]].

Sau đó, sử dụng hàm reshape để thay đổi hình dạng của array và tensor 3D vừa tạo có shape (3, 2, 3) thành array và tensor 2D có shape (3, 6).

3. Đáp án

Đáp án sẽ được gửi vào buổi tối trên nhóm.

Các Hàm Quan Trọng Trong Numpy, Pytorch, Tensorflow - Phần 3

Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Hàm **clip** được sử dụng để giới hạn giá trị của một array hoặc tensor để nằm trong một phạm vi được chỉ định. Các thư viện như NumPy, PyTorch và TensorFlow đều cung cấp hàm clip với cú pháp tương tự nhau.
Trong NumPy, hàm clip được sử dụng để giới giá trị của một mảng trong một khoảng được chỉ định. Cú pháp sử dụng:

```
np.clip(a, a_min, a_max)
```

Trong đó:

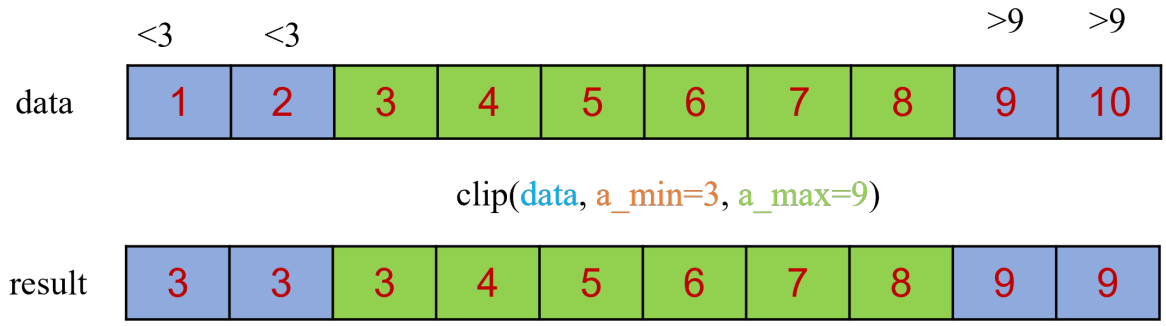
- a: Mảng đầu vào.
- a_min: Giá trị nhỏ nhất mà các phần tử của a được lấy.
- a_max: Giá trị lớn nhất mà các phần tử của a được lấy.

Ví dụ:

```
# Numpy code
import numpy as np

# Thiết lập seed để đảm bảo kết quả ngẫu nhiên có thể tái tạo được
np.random.seed(2024)

# Tạo một array với các giá trị ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 3)
arr_rand = np.random.randint(-10, 10, (3, 3))
print("Mảng ngẫu nhiên trong khoảng [-10, 10):\n", arr_rand)

# Sử dụng hàm clip để cắt các giá trị nằm ngoài khoảng [3, 8]
arr_clip = np.clip(arr_rand, a_min=3, a_max=8)
print("Mảng sau khi được cắt trong khoảng [3, 8]:\n", arr_clip)
```

```
================= Output ================
Mảng ngẫu nhiên trong khoảng [-10, 10):
 [[ -2 -10 -10]
 [ -6  -1  -9]
 [ -7   0  -8]]

Mảng sau khi được cắt trong khoảng [3, 8]:
 [[3 3 3]
 [3 3 3]
 [3 3 3]]
=========================================
```

Trong ví dụ trên, ta hiết lập **seed** giúp đảm bảo rằng kết quả ngẫu nhiên có thể được tái tạo. Tiếp theo, sử dụng **np.random.randint** để tạo một array với các giá trị ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 3). Sau đó **np.clip** được sử dụng để cắt các giá trị trong array sao cho chúng không vượt qua khoảng [3, 8]. Tức là giá trị nào nhỏ hơn 3 thì sẽ gán bằng 3, giá trị nào lớn hơn 8 sẽ gán bằng 8, các giá trị trong khoảng [3, 8] thì giữ nguyên.



Hình 1: Minh họa cách sử dụng hàm clip

Trong Pytorch, ta có thể sử dụng **clamp** hoặc **clip** đều được, vì trong Pytorch **clip** là tên bí danh hoặc có thể hiểu là tên viết tắt của **clamp**. Cú pháp:

```
# Pytorch
torch.clamp(input, min, max)
torch.clip(input, min, max)
```

Dưới đây là ví dụ:

```
#Pytorch code
import torch

# Thiết lập seed để đảm bảo kết quả ngẫu nhiên có thể tái tạo được
torch.manual_seed(2024)

# Tạo một tensor với các giá trị
# ngẫu nhiên trong khoảng [-10, 10) với
# kích thước (3, 4)
tensor_rand = torch.randint(-10, 10, size=(3, 4))
print("Tensor ngẫu nhiên trong khoảng [-10, 10):\n", tensor_rand)

# Sử dụng hàm clamp để cắt các giá trị nằm ngoài khoảng [3, 8]
tensor_clip = torch.clamp(tensor_rand, min=3, max=8)
print("Tensor sau khi được cắt trong khoảng [3, 8]:\n", tensor_clip)
```

```
================= Output ================
Tensor ngẫu nhiên trong khoảng [-10, 10):
 tensor([[  2,   0,  -6, -10],
        [ -7, -10,   0,   1],
        [  3,   9,   7,  -6]])

Tensor sau khi được cắt trong khoảng [3, 8]:
 tensor([[3, 3, 3, 3],
        [3, 3, 3, 3],
        [3, 8, 7, 3]])
=========================================
```

Trong Tensorflow, ta có thể sử dụng hàm **tf.clip_by_value** để giới hạn giá trị của tensor. Cú pháp:

```
#Tensorflow
tf.clip_by_value(t, clip_value_min, clip_value_max)
```

Dưới đây là ví dụ:

```

# Tensorflow code
import tensorflow as tf

# Thiết lập seed để đảm bảo kết quả ngẫu nhiên có thể tái tạo được
tf.random.set_seed(2024)

# Tạo một tensor với các giá trị
# ngẫu nhiên trong khoảng [-10, 10)
# với kích thước (3, 4)
tensor_random = tf.random.uniform((3, 4), minval=-10, maxval=10, dtype=tf.int32)
print("Tensor ngẫu nhiên trong khoảng [-10, 10):\n", tensor_random)

# Sử dụng hàm clip_by_value để cắt các giá trị nằm ngoài khoảng [3, 8]
tensor_clip = tf.clip_by_value(tensor_random, clip_value_min=3, clip_value_max=8)
print("Tensor sau khi được cắt trong khoảng [3, 8]:\n", tensor_clip)
```

```
================= Output ================
Tensor ngẫu nhiên trong khoảng [-10, 10):
 tf.Tensor(
[[-6 -2  0 -2]
 [ 4  4 -9  3]
 [ 7  1  2 -5]], shape=(3, 4),
 dtype=int32)

Tensor sau khi được cắt trong khoảng
[3, 8]:
 tf.Tensor(
[[3 3 3 3]
 [4 4 3 3]
 [7 3 3 3]], shape=(3, 4), dtype=int32)

=========================================
```

Hàm **clip** rất hữu ích khi ta muốn đảm bảo rằng các giá trị trong array hoặc tensor không vượt quá một khoảng cụ thể, giúp kiểm soát và chuẩn hóa dữ liệu, đặc biệt trong quá trình tiền xử lý dữ liệu cho mô hình máy học.

2. Bài tập

Câu 1: Viết chương trình tạo một Numpy array, Pytorch tensor, Tensorflow tensor với các giá trị số nguyên ngẫu nhiên trong khoảng [-5, 10) với kích thước (4, 3). Sử dụng hàm clip để cắt các giá trị nằm ngoài khoảng [-2, 0]? Lưu ý: Sử dụng seed=2024

```
# Numpy code
import numpy as np
np.random.seed(2024)

# Pytorch code
import torch
torch.manual_seed(2024)

# Tensorflow code
import tensorflow as tf
tf.random.set_seed(2024)
```

3. Đáp án

Đáp án sẽ được gửi vào buổi tối trên nhóm.

Indexing Trong Numpy, Pytorch và Tensorflow - Phần 1

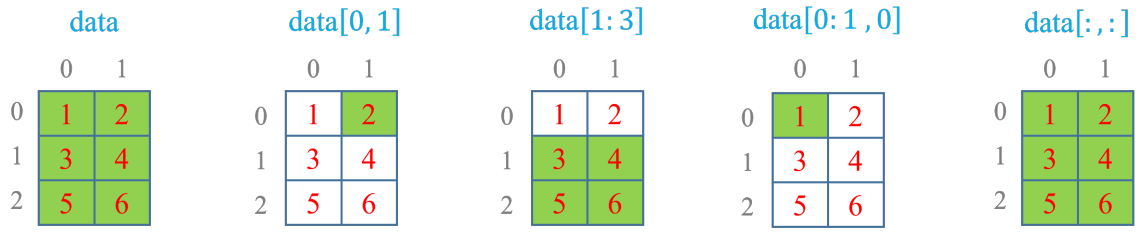
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

Slicing là một cách để trích xuất một phần của array, tensor dựa trên các chỉ số hoặc các điều kiện. Điều này cho phép ta lấy ra các phần tư theo các quy luật cụ thể, giúp thao tác dữ liệu dễ dàng hơn. Trong Numpy, cú pháp để thực hiện slicing là:

```
array[axis n row:column, axis n-1 row:column,...., axis 0 row:column,
      axis 1 row:column]
```

Trong đó: axis n row:column nghĩa là tại chiều thứ n, lấy phần tử từ hàng nào đến cột nào.



Hình 1: Minh họa cách sử dụng slicing

ví dụ:

```python
# Import thư viện Numpy
import numpy as np
# Tạo mảng 2D
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
# Slicing mảng tại row = 0, column = 1
# Giữ nguyên số chiều của mảng
sliced_array_1 = array_2d[0:1, 1:2]
print("sliced_array_1 (Giữ nguyên số chiều):")
print(sliced_array_1)
# Giảm số chiều của mảng
sliced_array_2 = array_2d[0, 1:2]
print("\nsliced_array_2 (Giảm số chiều):")
print(sliced_array_2)
# Giảm số chiều của mảng
sliced_array_3 = array_2d[0, 1]
print("\nsliced_array_3 (Giảm số chiều):")
print(sliced_array_3)
# Lấy toàn bộ mảng
sliced_array_4 = array_2d[:, :]
print("\nsliced_array_4 (Giữ nguyên số chiều):")
print(sliced_array_4)
```

```
================= Output ================
sliced_array_1 (Giữ nguyên số chiều):
[[2]]

sliced_array_2 (Giảm số chiều):
[2]

sliced_array_3 (Giảm số chiều):
2

sliced_array_4 (Giữ nguyên số chiều):
[[1 2 3]
 [4 5 6]]

=========================================
```

Trong ví dụ trên là các cách ta thực hiện slicing array, ở đây có hiện tượng giảm số chiều của mảng xuất hiện khi ta trích xuất một phần nhỏ của mảng thông qua slicing và chỉ giữ lại một số chiều của nó.
Ở trường hợp đầu tiên **sliced_array_1 = array_2d[0:1, 1:2]**, ta đang trích xuất một phần của mảng bằng cách chỉ định slicing cho cả hai chiều (hàng và cột). Kết quả là một mảng 2D với số chiều không thay đổi.
Trường hợp thứ hai **sliced_array_2 = array_2d[0, 1:2]** , ta đang trích xuất một hàng (0) và một phần của cột (từ 1 đến 1). Kết quả là một mảng 1D, và số chiều của mảng đã giảm xuống.
Tương tự như **sliced_array_2**, trường hợp **sliced_array_3 = array_2d[0, 1]** ta chỉ đang trích xuất một phần nhỏ của mảng, nên kết quả là một mảng 0D (véc-tơ). Trong Numpy, mảng 0D còn được gọi là scalar.
Trong trường hợp cuối, ta lấy toàn bộ mảng bằng cách không chỉ định slicing cho bất kỳ chiều nào cả. Kết quả là một mảng 2D và số chiều không thay đổi.
Quy tắc thực hiện slicing trong Numpy cũng áp dụng cho tensor trong Pytorch và Tensorflow. Ví dụ trong Pytorch:

```python
# Pytorch code
import torch


# Tạo tensor 2D
tensor_2d = torch.tensor([[1, 2, 3], [4,
    5, 6]])

# Slicing tensor tại vị trí row= 0,
# column = 1
# Giữ nguyên số chiều của tensor
sliced_tensor_1 = tensor_2d[0:1, 1:2]
print("sliced_tensor_1 (Giữ nguyên số
    chiều):")
print(sliced_tensor_1)

# Giảm số chiều của tensor
sliced_tensor_2 = tensor_2d[0, 1:2]
print("\nsliced_tensor_2 (Giảm số chiều):"
    )
print(sliced_tensor_2)

# Giảm số chiều của tensor
sliced_tensor_3 = tensor_2d[0, 1]
print("\nsliced_tensor_3 (Giảm số chiều):"
    )
print(sliced_tensor_3)

# Lấy toàn bộ tensor
sliced_tensor_4 = tensor_2d[:, :]
print("\nsliced_tensor_4 (Giữ nguyên số
    chiều):")
print(sliced_tensor_4)
```

```
================ Output ================
sliced_tensor_1 (Giữ nguyên số chiều):
tensor([[2]])

sliced_tensor_2 (Giảm số chiều):
tensor([2])

sliced_tensor_3 (Giảm số chiều):
tensor(2)

sliced_tensor_4 (Giữ nguyên số chiều):
tensor([[1, 2, 3],
        [4, 5, 6]])
========================================
```

Ví dụ trong Tensorflow:

```
# Tensorflow code
import tensorflow as tf
# Tạo tensor 2D
tensor_2d = tf.constant([[1, 2, 3], [4, 5, 6]])
# Slicing tensor tại vị trí row=0, column=1
# Giữ nguyên số chiều của tensor
sliced_tensor_1 = tensor_2d[0:1, 1:2]
print("sliced_tensor_1 (Giữ nguyên số chiều):")
print(sliced_tensor_1)
# Giảm số chiều của tensor
sliced_tensor_2 = tensor_2d[0, 1:2]
print("\nsliced_tensor_2 (Giảm số chiều):")
print(sliced_tensor_2)
# Giảm số chiều của tensor
sliced_tensor_3 = tensor_2d[0, 1]
print("\nsliced_tensor_3 (Giảm số chiều):")
print(sliced_tensor_3)
# Lấy toàn bộ tensor
sliced_tensor_4 = tensor_2d[:, :]
print("\nsliced_tensor_4 (Giữ nguyên số chiều):")
print(sliced_tensor_4)
```

```
================ Output ================
sliced_tensor_1 (Giữ nguyên số chiều):
tf.Tensor([[2]], shape=(1, 1), dtype=int32)

sliced_tensor_2 (Giảm số chiều):
tf.Tensor([2], shape=(1,), dtype=int32)

sliced_tensor_3 (Giảm số chiều):
tf.Tensor(2, shape=(), dtype=int32)

sliced_tensor_4 (Giữ nguyên số chiều):
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
========================================
```

Tùy thuộc vào mục đích mà ta có thể lựa chọn cách slice khác nhau.

2. Bài tập

Hãy viết chương trình để tạo một numpy array, PyTorch tensor và TensorFlow tensor từ một list 3D có giá trị như sau: [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]], [[13, 14, 15], [16, 17, 18]]]. Sau đó sử dụng slicing thực hiện các yêu cầu sau:

- Sử dụng slicing để lấy một phần của array, tensor tại vị trí index (0, 1) của mảng 2D trong mảng 3D và giữ nguyên số chiều
- Sử dụng slicing để lấy một phần của array, tensor tại vị trí index (0, 1) của mảng 2D trong mảng 3D, nhưng giảm đi một chiều.

3. Đáp án

Đáp án sẽ được gửi vào buổi tối trên nhóm.

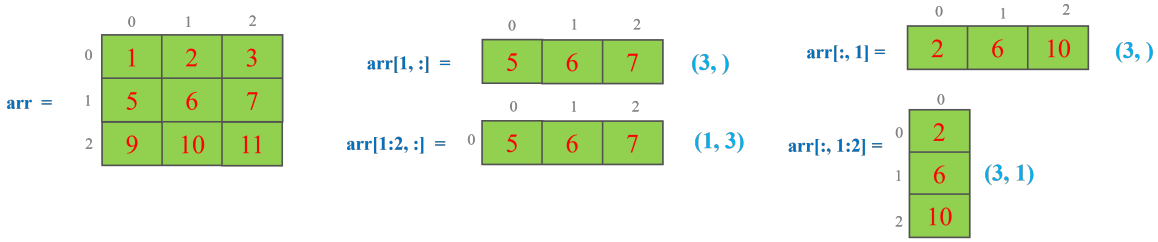
Indexing Trong Numpy, Pytorch và Tensorflow - Phần 2

Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

a) ***Truy xuất row, column***

Để truy cập một cột(column) hoặc một hàng(row) từ một array, tensor trong Numpy, Pytorch và Tensorflow có thể được thực hiện bằng cách sử dụng indexing.



Hình 1: Minh họa cách truy xuất row, column

Dưới đây là ví dụ và giải thích cách thực hiện:

```python
# Numpy code
import numpy as np

# Tạo mảng 2D
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Hai cách để lấy hàng index=1
# Cách 1: giảm số chiều của mảng
row_1_arr_1 = arr_2d[1, :]
print("row_1_arr_1 (Giảm số chiều):",
    row_1_arr_1)

# Cách 2: giữ nguyên số chiều của mảng
row_1_arr_2 = arr_2d[1:2, :]
print("row_1_arr_2 (Giữ nguyên số chiều):"
    , row_1_arr_2)

# Hai cách để lấy cột index=2
# Cách 1: giảm số chiều của mảng
column_2_arr_1 = arr_2d[:, 2]
print("\ncolumn_2_arr_1 (Giảm số chiều):\n
    ", column_2_arr_1)
# Cách 2: giữ nguyên số chiều của mảng
column_2_arr_2 = arr_2d[:, 2:3]
print("\ncolumn_2_arr_2
(Giữ nguyên số chiều):\n", column_2_arr_2)
```

```
================= Output ================
row_1_arr_1 (Giảm số chiều): [4 5 6]

row_1_arr_2 (Giữ nguyên số chiều): [[4 5
    6]]

column_2_arr_1 (Giảm số chiều):
 [3 6 9]

column_2_arr_2 (Giữ nguyên số chiều):
 [[3]
 [6]
 [9]]

=========================================
```

Ví dụ thư viện Pytorch, thực hiện tương tự như Numpy:

```python
# Pytorch code
import torch
# Tạo tensor 2D
tensor_2d = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Hai cách để lấy hàng index=1
# Cách 1: giảm số chiều của tensor
row_1_tensor_1 = tensor_2d[1, :]
print("row_1_tensor_1 (Giảm số chiều):", row_1_tensor_1)
# Cách 2: giữ nguyên số chiều của tensor
row_1_tensor_2 = tensor_2d[1:2, :]
print("row_1_tensor_2 (Giữ nguyên số chiều):", row_1_tensor_2)
# Hai cách để lấy cột index=2
# Cách 1: giảm số chiều của tensor
column_2_tensor_1 = tensor_2d[:, 2]
print("\ncolumn_2_tensor_1 (Giảm số chiều):\n", column_2_tensor_1)
# Cách 2: giữ nguyên số chiều của tensor
column_2_tensor_2 = tensor_2d[:, 2:3]
print("\ncolumn_2_tensor_2 (Giữ nguyên số chiều):\n", column_2_tensor_2)
```

```
================ Output ================
row_1_tensor_1 (Giảm số chiều):
tensor([4, 5, 6])

row_1_tensor_2 (Giữ nguyên số chiều):
tensor([[4, 5, 6]])

column_2_tensor_1 (Giảm số chiều):
 tensor([3, 6, 9])

column_2_tensor_2 (Giữ nguyên số chiều):
 tensor([[3],
         [6],
         [9]])
========================================
```

```python
# Tensorflow code
import tensorflow as tf
# Tạo tensor 2D
tensor_2d = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Hai cách để lấy hàng index=1
# Cách 1: giảm số chiều của tensor
row_1_tensor_1 = tensor_2d[1, :]
print("row_1_tensor_1 (Giảm số chiều):", row_1_tensor_1)
# Cách 2: giữ nguyên số chiều của tensor
row_1_tensor_2 = tensor_2d[1:2, :]
print("row_1_tensor_2 (Giữ nguyên số chiều):", row_1_tensor_2)
# Hai cách để lấy cột index=2
# Cách 1: giảm số chiều của tensor
column_2_tensor_1 = tensor_2d[:, 2]
print("\ncolumn_2_tensor_1 (Giảm số chiều):\n", column_2_tensor_1)
# Cách 2: giữ nguyên số chiều của tensor
column_2_tensor_2 = tensor_2d[:, 2:3]
print("\ncolumn_2_tensor_2 (Giữ nguyên số chiều):\n", column_2_tensor_2)
```

```
================ Output ================
row_1_tensor_1 (Giảm số chiều):
 tf.Tensor([4 5 6], shape=(3,), dtype=
   int32)

row_1_tensor_2 (Giữ nguyên số chiều):
 tf.Tensor([[4 5 6]], shape=(1, 3), dtype=
   int32)

column_2_tensor_1 (Giảm số chiều):
 tf.Tensor([3 6 9], shape=(3,), dtype=
   int32)

column_2_tensor_2 (Giữ nguyên số chiều):
 tf.Tensor(
[[3]
 [6]
 [9]], shape=(3, 1), dtype=int32)
========================================
```

Trong các ví dụ trên, chúng ta có một mảng, tensor hai chiều và chúng ta thực hiện một số thao tác để lấy dữ liệu từ hàng và cột cụ thể của mảng:

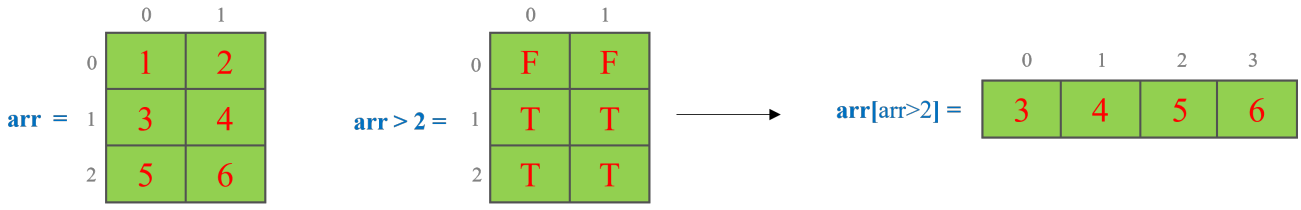
- Lấy hàng index=1: Chúng ta sử dụng hai cách khác nhau để lấy hàng thứ hai (‘[4, 5, 6]‘). Cách đầu tiên giảm số chiều của mảng, tensor, trong khi cách thứ hai giữ nguyên số chiều.

- Lấy cột index=2: Chúng ta sử dụng hai cách khác nhau để lấy cột thứ ba của mảng, tensor (‘[3, 6, 9]‘ hoặc ‘[[3], [6], [9]]‘). Cách đầu tiên giảm số chiều, trong khi cách thứ hai giữ nguyên số chiều bằng cách chỉ rõ là lấy từ vị trí bắt đầu và kết thúc của cột muốn lấy.

Tóm lại trong cả ba thư viện, ta có thể sử dụng phép indexing với dấu hai chấm (":") để chọn tất cả các phần tử trong một chiều của mảng hoặc tensor. Để truy cập một cột, ta giữ nguyên chiều hàng và chỉ định chỉ số của cột. Để truy cập một hàng, ta giữ nguyên chiều cột và chỉ định chỉ số của hàng.

b) ***Boolean indices***

Boolean indices (hoặc boolean indexing) là một cách để lấy hoặc lọc các phần tử từ một mảng dựa trên một mảng boolean có cùng hình dạng. Mảng boolean này được gọi là mask, và nó được sử dụng để chọn các phần tử tương ứng từ mảng gốc.



Hình 2: Minh họa cách sử dụng boolean indices

```python
# Import thư viện Numpy
import numpy as np

# Tạo mảng 2D
array_2d = np.array([[1, 2, 3], [4, 5, 6],[7, 8, 9]])

# Lấy các phần tử có giá trị lớn hơn 5
boolean_mask = array_2d > 5
filtered_result = array_2d[boolean_mask]

# In kết quả
print("Mảng 2D ban đầu:\n", array_2d)
print("\n Boolean mask:\n", boolean_mask)
print("\nCác phần tử lớn hơn 5:\n", filtered_result)
```

```
================= Output ================
Mảng 2D ban đầu:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

 Boolean mask:
 [[False False False]
 [False False  True]
 [ True  True  True]]

Các phần tử lớn hơn 5:
 [6 7 8 9]


=========================================
```

Trong ví dụ trên, **boolean_mask** được tạo ra bằng cách kiểm tra xem mỗi phần tử của arr có lớn hơn 5 hay không. Kết quả là một mảng **boolean_mask** với các giá trị True hoặc False. Sau đó, **boolean_mask** được sử dụng để lọc các phần tử tương ứng từ arr, chỉ giữ lại những phần tử có giá trị True.
Tương tự, cả Pytorch và Tensorflow cũng hỗ trợ boolean indexing. Quy trình là giống nhau như trong Numpy Ví dụ Pytorch:

```python
# Pytorch code
import torch

# Tạo tensor 2D
tensor_2d = torch.tensor([[1, 2, 3],
                          [4, 5, 6],
                          [7, 8, 9]])

# Lấy các phần tử có giá trị lớn hơn 5
boolean_mask = tensor_2d > 5
filtered_result = tensor_2d[boolean_mask]

# In kết quả
print("Tensor 2D ban đầu:\n", tensor_2d)
print("\n Boolean mask:\n", boolean_mask)
print("\nCác phần tử lớn hơn 5:\n",
    filtered_result)
```

```
================ Output ===============
Tensor 2D ban đầu:
 tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])

 Boolean mask:
 tensor([[False, False, False],
        [False, False,  True],
        [ True,  True,  True]])

Các phần tử lớn hơn 5:
 tensor([6, 7, 8, 9])


========================================
```

Ví dụ Tensorflow:

```python
# Tensorflow code
import tensorflow as tf


# Tạo tensor 2D
tensor_2d = tf.constant([[1, 2, 3],
                         [4, 5, 6],
                         [7, 8, 9]])

# Lấy các phần tử có giá trị lớn hơn 5
boolean_mask = tensor_2d > 5
filtered_result = tf.boolean_mask(
    tensor_2d, boolean_mask)

# In kết quả
print("Tensor 2D ban đầu:\n", tensor_2d)
print("\n Boolean mask:\n", boolean_mask)
print("\nCác phần tử lớn hơn 5:\n",
    filtered_result)
```

```
================ Output ===============
Tensor 2D ban đầu:
 tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]],
 shape=(3, 3), dtype=int32)

 Boolean mask:
 tf.Tensor(
[[False False False]
 [False False  True]
 [ True  True  True]],
 shape=(3, 3), dtype=bool)

Các phần tử lớn hơn 5:
 tf.Tensor([6 7 8 9],
 shape=(4,), dtype=int32)
========================================
```

2. Bài tập

Câu 1: Cho một list 2D như sau:

```
[[1, 0, 1],
 [0, 1, 0],
 [1, 0, 1]]
```

Hãy tạo array, tensor(Pytorch, Tensorflow) từ list 2D này sau đó thực hiện các yêu cầu sau:

- Lấy hàng thứ 2(index=1) theo 2 cách giữ nguyên hoặc giảm số chiều
- Lấy cột thứ nhất(index=0) theo 2 cách giữ nguyên hoặc giảm số chiều

Câu 2: Viết chương trình tạo một Numpy array, Pytorch tensor, Tensorflow tensor với các giá trị số nguyên ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 3). Sau đó sử dụng Boolean indices để lọc các phần tử lớn hơn 0. Lưu ý: sử dụng seed=2024

3. Đáp án

Đáp án sẽ được gửi vào buổi tối trên nhóm.

Các Phép Tính Numpy, Pytorch và Tensorflow - Phần 1

Dinh-Tiem Nguyen và Quang-Vinh Dinh

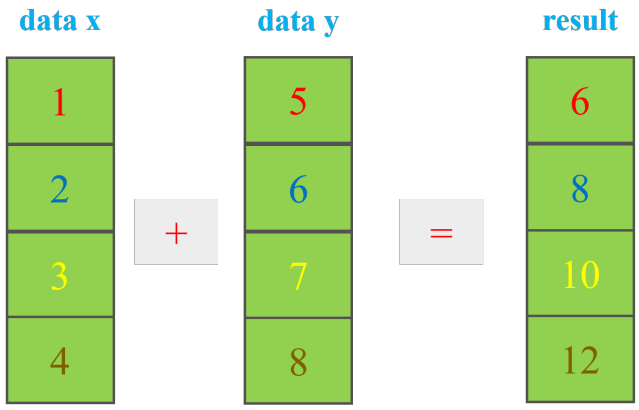
1. Mô tả

Trong phần này, chúng ta sẽ tìm hiểu về các phép tính cơ bản nhưng thất sự quan trong khi làm việc với array, tensor. Chúng ta sẽ tạm thời quên đi vòng for và sẽ thấy được sức mạnh tính toán của những thư viện này thật sự kinh ngạc.

a) ***Addition***

Chúng ta đã quá quen với các phép tính cộng, trừ, nhân, chia vì chúng ta sử dụng nó hằng ngày. Chúng ta biết về cách tính tuần tự, và gặp khó khăn khi đối mặt với số lượng tính toán lớn. Thư viện Numpy, Pytorch, Tensorflow được xây dựng để giải quyết vấn đề đó, chúng hỗ trợ việc tính toán song song, tức là thực hiện nhiều phép tính đồng thời.
Trước khi bắt đầu với các ví dụ thì ta hãy xem lại định nghĩa về phép cộng: Cho hai số a và b, phép cộng được biểu diễn bằng ký hiệu '+' và kết quả của phép cộng là tổng của a và b, thường được ký hiệu là $a + b$.
Đối với array và tensor thì phép cộng giữa chúng được thực hiện theo cùng một nguyên tắc cơ bản, là phép cộng element-wise (theo từng phần tử). Phép cộng này được định nghĩa như sau: Cho hai array (hoặc tensor) A và B cùng kích thước, phép cộng giữa chúng $(A + B)$ tạo ra một array mới C, với mỗi phần tử C_i được tính bằng cách cộng phần tử A_i với phần tử B_i.

$$C_i = A_i + B_i$$



Hình 1: Minh họa hàm addition

Trong các ví dụ dưới đây, ta thực hiện phép cộng các array, tensor bằng cách sử dụng toán tử "+"hoặc hàm add().
Ví dụ với Numpy, phép cộng giữa hai array có cùng kích thước là việc thực hiện phép cộng giữa các phần tử tương ứng của chúng.

```python
# Numpy code
import numpy as np
# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
# Thực hiện phép cộng hai array
# Cách 1: Sử dụng toán tử '+'
arr_add_1 = arr_1 + arr_2
# Cách 2: Sử dụng hàm np.add()
arr_add_2 = np.add(arr_1, arr_2)
# In ra màn hình
print(f"arr_1 = \n{arr_1}")
print(f"arr_2 = \n{arr_2}")
print("arr_1 + arr_2")
print(f"arr_add_1 = \n{arr_add_1}")
print(f"arr_add_2 = \n{arr_add_2}")
```

```
================ Output ===============
arr_1 =
[1 2 3 4]

arr_2 =
[5 6 7 8]

arr_1 + arr_2

arr_add_1 =
[ 6  8 10 12]

arr_add_2 =
[ 6  8 10 12]

=======================================
```

Kết quả thực hiện phép cộng sẽ là một array mới là [6, 8, 10, 12] với mỗi phần tử là tổng của các phần tử tương ứng trong arr1 và arr2.
Tương tự, chúng ta có thể sử dụng toán tử "+"và hàm add() trong thư viện Pytorch và Tensorflow:

```python
#Pytorch code
import torch
# Tạo hai tensor 1D
tensor_1 = torch.tensor([1, 2, 3, 4])
tensor_2 = torch.tensor([5, 6, 7, 8])
# Thực hiện phép cộng hai tensor
# Cách 1: Sử dụng toán tử '+'
tensor_add_1 =  tensor_1 + tensor_2
# Cách 2: Sử dụng hàm torch.add()
tensor_add_2 = torch.add(tensor_1,
    tensor_2)
# In ra màn hình
print(f"tensor_1 = \n{tensor_1}")
print(f"tensor_2 = \n{tensor_2}")
print(f"tensor_1 + tensor_2")
print(f"tensor_add_1 = \n{tensor_add_1}")
print(f"tensor_add_2 = \n{tensor_add_2}")
```

```
================ Output ===============
tensor_1 =
tensor([1, 2, 3, 4])

tensor_2 =
tensor([5, 6, 7, 8])

tensor_1 + tensor_2

tensor_add_1 =
tensor([ 6,  8, 10, 12])

tensor_add_2 =
tensor([ 6,  8, 10, 12])


=======================================
```

```python
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 1D
tensor_1 = tf.constant([1, 2, 3, 4])
tensor_2 = tf.constant([5, 6, 7, 8])

# Thực hiện phép cộng hai tensor
# Cách 1: Sử dụng toán tử '+'
tensor_add_1 =  tensor_1 + tensor_2
# Cách 2: Sử dụng hàm tf.add()
tensor_add_2 = tf.add(tensor_1, tensor_2)

print(f"tensor_1 = \n{tensor_1}")
print(f"tensor_2 = \n{tensor_2}")
print(f"tensor_1 + tensor_2")
print(f"tensor_add_1 = \n{tensor_add_1}")
print(f"tensor_add_2 = \n{tensor_add_2}")
```

```
================ Output ===============
tensor_1 =
[1 2 3 4]

tensor_2 =
[5 6 7 8]

tensor_1 + tensor_2

tensor_add_1 =
[ 6  8 10 12]

tensor_add_2 =
[ 6  8 10 12]



=======================================
```

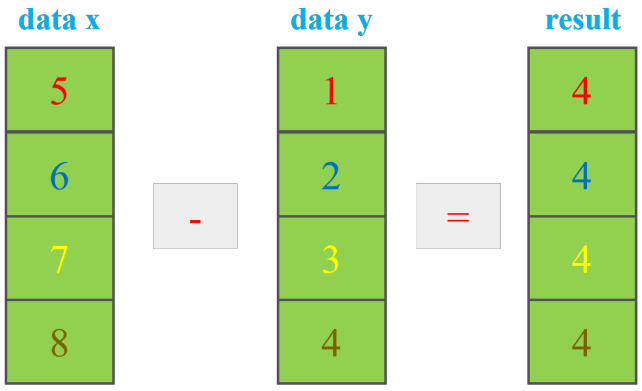
Từ các ví dụ trên, ta có thể thấy phép cộng array, tensor có tính chất sau: Phép cộng array, tensor là một phép toán element-wise, tức là mỗi phần tử trong kết quả được tính toán dựa trên các phần tử tương ứng của các array, tensor gốc. Điều này giúp thực hiện các phép toán một cách hiệu quả trên toàn bộ array, tensor, làm giảm sự cần thiết phải sử dụng vòng lặp.

b) ***Subtraction***

Trong toán học, phép trừ hai số được định nghĩa như sau: Giả sử có hai số a và b, thì $a - b$ là kết quả của phép trừ. Kết quả này cho biết khoảng cách giữa a và b trên trục số.
Đối với phép trừ hai array hoặc tensor cùng kích thước A và B, phép trừ giữa chúng $(A - B)$ tạo ra một array mới C, với mỗi phần tử C_i được tính bằng cách trừ phần tử B_i khỏi phần tử A_i.

$$C_i = A_i - B_i$$

Tương tự như phép cộng array, tensor, phép trừ cũng là một phép toán element-wise, tức là mỗi phần tử của array kết quả được tính bằng cách trừ phần tử tương ứng của array thứ nhất cho phần tử tương ứng của array thứ hai.



Hình 2: Minh họa hàm subtraction

Ví dụ phép trừ hai array trong Numpy, chúng ta có thể sử dụng toán tử "-"hoặc hàm **np.subtract()**

```python
# Numpy code
import numpy as np
# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
print("arr_1:", arr_1)
print("arr_2:", arr_2)
# Hiệu hai array
# Cách 1: Sử dụng toán tử '-'
array_sub_1 = arr_1 - arr_2
# Cách 2: Sử dụng hàm np.subtract()
array_sub_2 = np.subtract(arr_1, arr_2)
# In ra màn hình
print("Cách 1\n", array_sub_1)
print("Cách 2\n", array_sub_2)
```

```
================= Output ================
arr_1: [1 2 3 4]

arr_2: [5 6 7 8]

Cách 1
 [-4 -4 -4 -4]

Cách 2
 [-4 -4 -4 -4]

=========================================
```

Đối với Pytorch, ta dùng **torch.sub()** hoặc **torch.subtract()**.

```pytorch
# Pytorch code
import torch
# Tạo hai tensor 1D
tensor_1 = torch.tensor([1, 2, 3, 4])
tensor_2 = torch.tensor([5, 6, 7, 8])
# In ra giá trị của hai tensor
print("tensor 1:", tensor_1)
print("tensor 2:", tensor_2)
# Hiệu hai tensor
# Cách 1: Sử dụng toán tử '-'
tensor_sub_1 = tensor_1 - tensor_2
# Cách 2: Sử dụng hàm torch.sub()
tensor_sub_2 = torch.sub(tensor_1,
    tensor_2)
# In ra màn hình
print("Cách 1\n", tensor_sub_1)
print("Cách 2\n", tensor_sub_2)
```

```output
================ Output ===============
tensor 1: tensor([1, 2, 3, 4])

tensor 2: tensor([5, 6, 7, 8])

Cách 1
 tensor([-4, -4, -4, -4])

Cách 2
 tensor([-4, -4, -4, -4])

=======================================
```

```tensorflow
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 1D
tensor_1 = tf.constant([1, 2, 3, 4])
tensor_2 = tf.constant([5, 6, 7, 8])

# In ra giá trị của hai tensor
print("tensor 1:", tensor_1)
print("tensor 2:", tensor_2)

# Hiệu hai tensor
# Cách 1: Sử dụng toán tử '-'
tensor_sub_1 = tensor_1 - tensor_2

# Cách 2: Sử dụng hàm tf.subtract()
tensor_sub_2 = tf.subtract(tensor_1,
    tensor_2)

# In ra màn hình
print("Cách 1\n", tensor_sub_1)
print("Cách 2\n", tensor_sub_2)
```

```output
================ Output ===============
tensor 1: tf.Tensor([1 2 3 4], shape=(4,),
    dtype=int32)

tensor 2: tf.Tensor([5 6 7 8], shape=(4,),
    dtype=int32)

Cách 1
 tf.Tensor([-4 -4 -4 -4], shape=(4,),
   dtype=int32)

Cách 2
 tf.Tensor([-4 -4 -4 -4], shape=(4,),
   dtype=int32)

=======================================
```

2. Bài tập

Cho hai list 2D như sau:

```
lst_1 = [[1, -2, 1],
         [-3, 1, 0],
         [-2, 5, 1]]

lst_2 = [[1, 3, 5],
         [2, 4, 6],
         [3, 5, 7]]
```

Hãy tạo 2 array, tensor(Pytorch, Tensorflow) từ list 2D này sau đó thực hiện các phép tính addition và subtraction giữa hai array, tensor vừa tạo.

3. Đáp án

Đáp án sẽ được cập nhật vào buổi tối trong nhóm!

Các Phép Tính Numpy, Pytorch và Tensorflow - Phần 2

Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

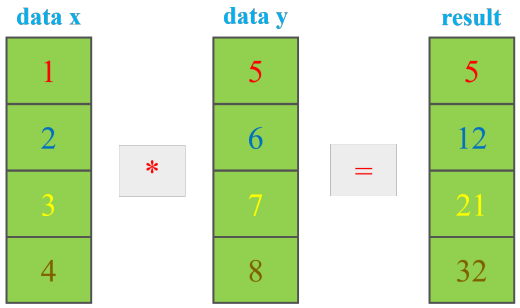
Trong phần này, chúng ta sẽ tiếp tục tìm hiểu về các phép tính cơ bản nhưng thất sự quan trong khi làm việc với array, tensor.

a) ***Multiplication***

Phép nhân **Multiplication** array, tensor được thực hiện theo nguyên tắc element-wise: Cho hai array (hoặc tensor) cùng kích thước A và B, phép nhân giữa chúng $(A \cdot B)$ tạo ra một array (hoặc tensor) mới C, với mỗi phần tử C_{ij} được tính bằng cách nhân phần tử A_{ij} với phần tử B_{ij}.

$$C_{ij} = A_{ij} \cdot B_{ij}$$

Để thực hiện multiplication, ta sử dụng toán tử "*"hoặc hàm **multiply**.



Hình 1: Minh họa phép nhân multiplication

Ví dụ với thư viện Numpy:

```python
# Numpy code
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Tích hai array
# Cách 1: Sử dụng toán tử '*'
result_mul_1 = arr_1 * arr_2
# Cách 2: Sử dụng hàm np.multiply()
result_mul_2 = np.multiply(arr_1, arr_2)
# In ra màn hình
print("Cách 1:\n", result_mul_1)
print("Cách 2:\n", result_mul_2)
```

```
================= Output ================
arr_1:
 [1 2 3 4]

arr_2:
 [5 6 7 8]

Cách 1:
 [ 5 12 21 32]

Cách 2:
 [ 5 12 21 32]

=========================================
```

Ví dụ với thư viện Pytorch

```python
# Pytorch code
import torch

# Tạo hai tensor 1D
tensor_1 = torch.tensor([1, 2, 3, 4])
tensor_2 = torch.tensor([5, 6, 7, 8])

# In ra giá trị của hai tensor
print("tensor 1:", tensor_1)
print("tensor 2:", tensor_2)

# Tích hai tensor
# Cách 1: Sử dụng toán tử '*'
result_mul_1 = tensor_1 * tensor_2

# Cách 2: Sử dụng hàm torch.multiply()
result_mul_2 = torch.multiply(tensor_1,
    tensor_2)

# In ra màn hình
print("Cách 1\n", result_mul_1)
print("Cách 2\n", result_mul_2)
```

```
================ Output ===============
tensor 1: tensor([1, 2, 3, 4])

tensor 2: tensor([5, 6, 7, 8])

Cách 1
 tensor([ 5, 12, 21, 32])

Cách 2
 tensor([ 5, 12, 21, 32])

=======================================
```

Ví dụ với thư viện Tensorflow

```python
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 1D
tensor_1 = tf.constant([1, 2, 3, 4])
tensor_2 = tf.constant([5, 6, 7, 8])

# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)

# Tích hai tensor
# Cách 1: Sử dụng toán tử '*'
result_mul_1 = tensor_1 * tensor_2

# Cách 2: Sử dụng hàm tf.multiply()
result_mul_2 = tf.multiply(tensor_1,
    tensor_2)

# In ra màn hình
print("Cách 1:\n", result_mul_1)
print("Cách 2:\n", result_mul_2)
```

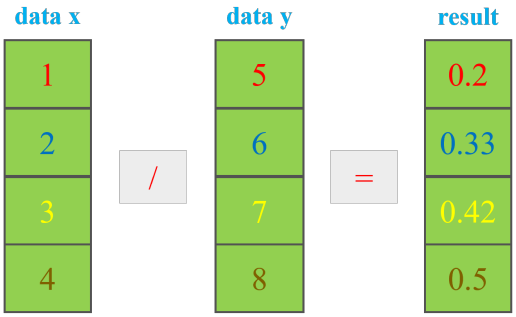
```
================ Output ===============
tensor 1:
 tf.Tensor([1 2 3 4], shape=(4,), dtype=
    int32)

tensor 2:
 tf.Tensor([5 6 7 8], shape=(4,), dtype=
    int32)

Cách 1:
 tf.Tensor([ 5 12 21 32], shape=(4,),
    dtype=int32)

Cách 2:
 tf.Tensor([ 5 12 21 32], shape=(4,),
    dtype=int32)

=======================================
```

b) ***Division***

Phép chia (division) array và tensor được thực hiện theo nguyên tắc element-wise. Cho hai array (hoặc tensor) cùng kích thước A và B, phép chia giữa chúng ($A \div B$) tạo ra một array (hoặc tensor) mới C, với mỗi phần tử C_{ij} được tính bằng cách chia phần tử A_{ij} cho phần tử B_{ij}.

$$C_{ij} = \frac{A_{ij}}{B_{ij}}$$

Để thực hiện phép chia array, tensor, ta có thể sử dụng toán tử "/"hoặc hàm divide().



Hình 2: Minh họa phép chia Division

Ví dụ với thư viện Numpy:

```python
# Numpy code
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2, 3, 4])
arr_2 = np.array([5, 6, 7, 8])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Chia hai array
# Cách 1: Sử dụng toán tử '/'
result_div_1 = arr_1 / arr_2
# Cách 2: Sử dụng hàm np.divide()
result_div_2 = np.divide(arr_1, arr_2)
# In ra màn hình
print("Cách 1:\n", result_div_1)
print("Cách 2:\n", result_div_2)
```

```
================ Output ===============
arr_1:
 [1 2 3 4]

arr_2:
 [5 6 7 8]

Cách 1:
 [0.2  0.33333333 0.42857143 0.5]

Cách 2:
 [0.2  0.33333333 0.42857143 0.5]

=======================================
```

Ví dụ với thư viện Pytorch:

```python
# Pytorch code
import torch

# Tạo hai tensor 1D
tensor_1 = torch.tensor([1, 2, 3, 4])
tensor_2 = torch.tensor([5, 6, 7, 8])
# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)
# Chia hai tensor
# Cách 1: Sử dụng toán tử '/'
result_div_1 = tensor_1 / tensor_2
# Cách 2: Sử dụng hàm torch.div()
result_div_2 = torch.divide(tensor_1,
    tensor_2)
# In ra màn hình
print("Cách 1:\n", result_div_1)
print("Cách 2:\n", result_div_2)
```

```
================ Output ===============
tensor 1:
 tensor([1, 2, 3, 4])

tensor 2:
 tensor([5, 6, 7, 8])

Cách 1:
 tensor([0.2000, 0.3333, 0.4286, 0.5000])

Cách 2:
 tensor([0.2000, 0.3333, 0.4286, 0.5000])

=======================================
```

Ví dụ với thư viện Tensorflow:

```
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 1D
tensor_1 = tf.constant([1, 2, 3, 4])
tensor_2 = tf.constant([5, 6, 7, 8])
# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)
# Chia hai tensor
# Cách 1: Sử dụng toán tử '/'
result_div_1 = tensor_1 / tensor_2
# Cách 2: Sử dụng hàm tf.divide()
result_div_2 = tf.divide(tensor_1,
    tensor_2)
# In ra màn hình
print("Cách 1:\n", result_div_1)
print("Cách 2:\n", result_div_2)
```

```
================ Output ===============
tensor 1:
 tf.Tensor([1 2 3 4], shape=(4,),
 dtype=int32)

tensor 2:
 tf.Tensor([5 6 7 8], shape=(4,),
 dtype=int32)

Cách 1:
 tf.Tensor([0.2 0.33333333 0.42857143
    0.5],
 shape=(4,), dtype=float64)
Cách 2:
 tf.Tensor([0.2 0.33333333 0.42857143
    0.5],
 shape=(4,), dtype=float64)
=======================================
```

2. Bài tập Cho hai list 2D như sau:

```
lst_1 = [[1, -2, 1],
         [-3, 1, 4],
         [-2, 5, 1]]

lst_2 = [[1, 3, 5],
         [2, 4, 6],
         [3, 5, 7]]
```

Hãy tạo 2 array, tensor(Pytorch, Tensorflow) từ list 2D này sau đó thực hiện các phép tính multiplication và division giữa hai array, tensor vừa tạo.

3. Đáp án

Các Phép Tính Numpy, Pytorch và Tensorflow - Phần 3

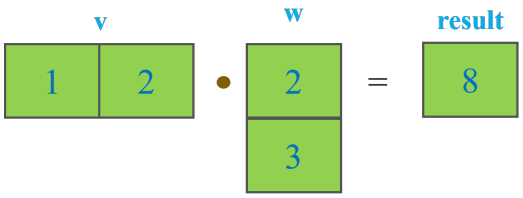
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

a) ***Inner product***

Phép tính tích vô hướng (inner product), còn được biết đến với tên gọi khác là dot product. Phép tích vô hướng giữa hai vector có thể được tính để đo lường sự tương quan hoặc hướng của chúng.
Cho hai vectơ $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ và $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ trong không gian Euclidean $\mathbb{R}^n$, phép tích vô hướng của chúng được xác định bởi công thức:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$



Hình 1: Minh họa inner

Trong thư viện Numpy, hàm **np.dot()** được sử dụng để thực hiện phép nhân ma trận, và nó cũng hỗ trợ tích vô hướng cho các vectơ(array 1D). Cú pháp:

```
np.dot(a, b)
```

Trong đó a, b là các array để thực hiện phép nhân. Các chiều của a và b phải thích hợp để thực hiện phép nhân (ví dụ, số cột của a phải bằng số hàng của b).

```
# Numpy code
import numpy as np

# Tạo hai array 1D
arr_1 = np.array([1, 2])
arr_2 = np.array([2, 3])
# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)
# Sử dụng hàm np.dot()
result_dot = np.dot(arr_1, arr_2)
# In ra màn hình
print("Kết quả tích vô hướng arr_1, arr_2
    :\n", result_dot)
```

```
================ Output ================
arr_1:
 [1 2]

arr_2:
 [2 3]

Kết quả tích vô hướng arr_1, arr_2:
 8

========================================
```

Trong Pytorch, hàm **torch.dot()** được sử dụng để thực hiện tích vô hướng (dot product) giữa hai vectơ. Tuy nhiên khác với Numpy có thể tính tích của các array nhiều chiều thì hàm này chỉ cho phép tính tích vô hướng của hai vector 1D. Cú pháp:

```
torch.dot(tensor1, tensor2)
```

Trong đó, tensor1, tensor2: Hai tensor có cùng kích thước để thực hiện tích vô hướng. Kết quả trả về sẽ là một số vô hướng.

```
# Pytorch code
import torch

# Tạo hai tensor 1D
tensor_1 = torch.tensor([1, 2])
tensor_2 = torch.tensor([2, 3])
# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)
# Sử dụng hàm torch.dot()
result_dot = torch.dot(tensor_1, tensor_2)
# In ra màn hình
print("Kết quả tích vô hướng tensor_1,
    tensor_2:\n", result_dot)
```

```
================ Output ===============
arr_1:
 [1 2]

arr_2:
 [2 3]

Kết quả tích vô hướng arr_1, arr_2:
 8

=======================================
```

Trong Tensorflow, hàm **tf.tensordot()** được sử dụng để thực hiện phép nhân tensor với nhau theo các chiều chỉ định. Nó có thể được sử dụng để thực hiện nhiều loại phép nhân tensor, bao gồm cả phép nhân ma trận và tích vô hướng. Cú pháp:

```
tf.tensordot(a, b, axes)
```

Trong đó, các tham số được định nghĩa như sau:

- a, b: Hai tensor để thực hiện phép nhân.
- axes: Một tuple hoặc số nguyên chỉ định các chiều của a và b được sử dụng trong phép nhân. Nếu là một số nguyên, nó xác định số chiều được sử dụng trong cả a và b.

```
# TensorFlow code
import tensorflow as tf

# Tạo hai tensor 1D
tensor_1 = tf.constant([1, 2])
tensor_2 = tf.constant([2, 3])

# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)

# Tích vô hướng hai tensor
result_dot = tf.tensordot(tensor_1,
    tensor_2, axes=1)

# In ra màn hình
print("Kết quả tích vô hướng tensor_1,
    tensor_2:\n", result_dot)
```

```
================ Output ===============
arr_1:
 [1 2]

arr_2:
 [2 3]

Kết quả tích vô hướng arr_1, arr_2:
 8

=======================================
```

Tóm lại thì khi thực hiện tính tích vô hướng hai vector thì chúng ta dùng dot product. Với thư viện Numpy và Tensorflow thì chúng ta có thể áp dụng dot product với array, tensor nhiều chiều phức tạp hơn.

b) ***Matrix-matrix multiplication***
Trong toán học, phép nhân ma trận (matrix multiplication) thường được biểu diễn bằng dấu chấm $(\cdot)$ hoặc dấu nhân $(\times)$. Các ma trận A và B có thể được nhân với nhau để tạo ra một ma trận kết quả C, với điều kiện số cột của ma trận A bằng số hàng của ma trận B.
Giả sử A là một ma trận có kích thước $m \times n$ (m hàng, n cột), và B là một ma trận có kích thước $n \times p$ (n hàng, p cột), thì ma trận kết quả C sẽ có kích thước $m \times p$. Quy tắc này còn được biết đến như là quy tắc hàng-cột, vì mỗi phần tử của ma trận kết quả C được tính bằng cách lấy tổng của tích từng phần tử của hàng tương ứng của ma trận A và cột tương ứng của ma trận B.
Phép nhân ma trận $C = A \times B$ có thể được mô tả như sau:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

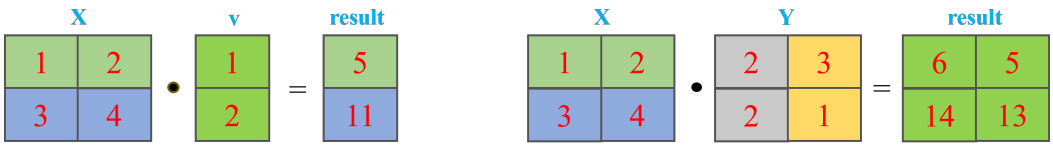
Trong đó:

- C_{ij} là phần tử ở hàng thứ i và cột thứ j của ma trận C.
- A_{ik} là phần tử ở hàng thứ i và cột thứ k của ma trận A.
- B_{kj} là phần tử ở hàng thứ k và cột thứ j của ma trận B.
- Phép toán $\sum_{k=1}^{n}$ biểu thị việc tính tổng qua tất cả các giá trị của k từ 1 đến n.

Ví dụ:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Trong trường hợp này, $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$, $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$, và tương tự cho các phần tử còn lại của ma trận C.



Hình 2: Minh họa phép nhân ma trận

Trong thư viện Numpy, hàm **np.matmul()** được sử dụng để thực hiện phép nhân ma trận. Hàm này cung cấp một cách linh hoạt để thực hiện nhân ma trận giữa các mảng (hoặc tensor) Numpy. Sử dụng cú pháp:

```
np.matmul(a, b, out=None)
```

Trong đó các tham số được định nghĩa như sau:

- a, b: Các ma trận (hoặc mảng) để thực hiện phép nhân. Số cột của ma trận a phải bằng số hàng của ma trận b.
- out: Mảng kết quả. Nếu được chỉ định, kết quả sẽ được lưu trữ vào mảng này.

```python
# Numpy code
import numpy as np

# Tạo 2 array 2D
arr_1 = np.array([[1, 2], [3, 4]])
arr_2 = np.array([[5, 6], [7, 8]])

# In ra giá trị của hai array
print("arr_1:\n", arr_1)
print("arr_2:\n", arr_2)

# Sử dụng hàm np.matmul()
result_matmul = np.matmul(arr_1, arr_2)

# In ra màn hình
print("Kết quả tích arr_1, arr_2:\n",
    result_matmul)
```

```
================ Output ===============
arr_1:
 [[1 2]
 [3 4]]

arr_2:
 [[5 6]
 [7 8]]

Kết quả tích arr_1, arr_2:
 [[19 22]
 [43 50]]


=======================================
```

Ví dụ PytorchPytorch:

```python
# Pytorch code
import torch

# Tạo hai tensor 2D
tensor_1 = torch.tensor([[1, 2], [3, 4]])
tensor_2 = torch.tensor([[5, 6], [7, 8]])

# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)

# Sử dụng hàm torch.matmul()
result_matmul = torch.matmul(tensor_1,
    tensor_2)

# In ra màn hình
print("Kết quả tích tensor_1, tensor_2:\n"
    , result_matmul)
```

```
================ Output ===============
tensor 1:
 tensor([[1, 2],
        [3, 4]])

tensor 2:
 tensor([[5, 6],
        [7, 8]])

Kết quả tích tensor_1, tensor_2:
 tensor([[19, 22],
        [43, 50]])


=======================================
```

Trong Pytorch, hàm **torch.matmul()** được sử dụng để thực hiện phép nhân ma trận (matrix multiplication) giữa hai tensor. Hàm này hỗ trợ cả tính tích vô hướng cho vectơ và cũng có thể được sử dụng cho các trường hợp phức tạp hơn. Cú pháp sử dụng là:

```python
torch.matmul(input, other, out=None)
```

Hàm này sẽ trả về kết quả của phép nhân ma trận, trong đó các tham số được định nghĩa như sau:

- input: Tensor đầu tiên để thực hiện phép nhân ma trận.
- other: Tensor thứ hai để thực hiện phép nhân ma trận.
- out: Tensor để lưu trữ kết quả. Nếu không được chỉ định, một tensor mới sẽ được tạo.

Hàm **tf.matmul()** trong Tensorflow được sử dụng để thực hiện phép nhân ma trận giữa hai tensor. Nó hỗ trợ cả việc thực hiện phép nhân ma trận và tích vô hướng cho các tensor có số chiều khác nhau. Cú pháp:

```python
tf.matmul(a, b, transpose_a=False, transpose_b=False)
```

Hàm này trả về kết quả là phép nhân ma trận, trong đó các tham số được định nghĩa như sau:

- a, b: Hai tensor để thực hiện phép nhân ma trận.
- transpose_a, transpose_b: Xác định xem có nên chuyển vị các tensor a và b trước khi thực hiện phép nhân hay không.

Ví dụ:

```
# Tensorflow code
import tensorflow as tf

# Tạo hai tensor 2D
tensor_1 = tf.constant([[1, 2], [3, 4]])
tensor_2 = tf.constant([[5, 6], [7, 8]])

# In ra giá trị của hai tensor
print("tensor 1:\n", tensor_1)
print("tensor 2:\n", tensor_2)

# Tích hai tensor
result_matmul = tf.matmul(tensor_1,
    tensor_2)

# In ra màn hình
print("Kết quả tích tensor_1, tensor_2:\n"
    , result_matmul)
```

```
================ Output ================
tensor 1:
 tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tensor 2:
 tf.Tensor(
[[5 6]
 [7 8]], shape=(2, 2), dtype=int32)

Kết quả tích tensor_1, tensor_2:
 tf.Tensor(
[[19 22]
 [43 50]], shape=(2, 2), dtype=int32)

========================================
```

Mặc dù hàm matmul khá giống với dot product ở phần trước, nhưng hai hàm này là khác nhau nên ta cần chú ý khi sử dụng chúng.

2. Bài tập

Câu 1: Cho hai vector a = [1, 4, 7], b = [9, 2, 3] Tính tích vô hướng của hai vector này bằng ba thư viện Numpy, Pytorch về Tensorfow

Câu 2: Viết chương trình tạo hai Numpy array, Pytorch tensor, Tensorflow tensor với các giá trị số nguyên ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 3). Hãy tính matrix multiplication hai ma trận này bằng ba thư viện Numpy, Pytorch và Tensorflow. Lưu ý: sử dụng seed=2024

3. Đáp án

Các Phép Tính Numpy, Pytorch và Tensorflow Transpose và Summation

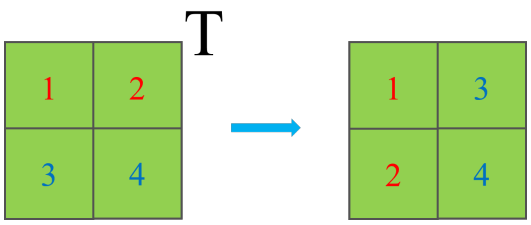
Dinh-Tiem Nguyen và Quang-Vinh Dinh

1. Mô tả

a) ***Chuyển vị - Transpose***

Chuyển vị (Transpose) là một phép toán quan trọng trong đại số tuyến tính, nó thực hiện thay đổi vị trí của các hàng thành cột và ngược lại trong một ma trận. Kí hiệu của phép toán chuyển vị thường được biểu diễn bằng kí hiệu ^{T} hoặc là $^\intercal$. Nếu A là một ma trận, thì chuyển vị của A được kí hiệu là A^T hoặc $A^\intercal$.
Công Thức Chuyển Vị: Nếu A là một ma trận với các phần tử a_{ij}, thì chuyển vị của A, ký hiệu là A^T hay $A^\intercal$, có kích thước là số cột của A trở thành số hàng của A và ngược lại. Cụ thể, nếu A có kích thước $m \times n$, thì A^T có kích thước $n \times m$.

$$(A^T)_{ij} = A_{ji}$$



Hình 1: Minh họa transpose

Ví dụ:
Giả sử có ma trận A như sau:

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

Chuyển vị của A, ký hiệu là A^T, sẽ là:

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Một số tính chất của phép chuyển vị:

- Chuyển vị của một ma trận chuyển vị lại sẽ cho ra ma trận ban đầu: $(A^T)^T = A$.
- Chuyển vị của tổng hai ma trận bằng tổng chuyển vị của từng ma trận: $(A + B)^T = A^T + B^T$.
- Chuyển vị của tích hai ma trận bằng tích đảo ngược vị trí của chuyển vị từng ma trận: $(AB)^T = B^T A^T$.

Ví dụ:

```python
# Numpy code
import numpy as np

#Tạo array 2d
arr_1 = np.array([[1, 2], [3, 4]])
# Chuyển vị array
# Cách 1: Sử dụng hàm np.transpose()
arr_transposed_1 = np.transpose(arr_1)
# Cách 2: Sử dụng toán tử T
arr_transposed_2 = arr_1.T
# In ra màn hình
print("array 1:\n", arr_1)
print("array 1 sau khi chuyển vị, cách 1:\
    n", arr_transposed_1)
print("array 1 sau khi chuyển vị, cách 2:\
    n", arr_transposed_2)
```

```
================ Output ===============
array 1:
 [[1 2]
 [3 4]]

array 1 sau khi chuyển vị, cách 1:
 [[1 3]
 [2 4]]

array 1 sau khi chuyển vị, cách 2:
 [[1 3]
 [2 4]]



=======================================
```

Trong Numpy, chuyển vị của một mảng (array) có thể được thực hiện bằng cách sử dụng hàm **np.transpose()** hoặc toán tử chuyển vị **.T**. Chuyển vị thay đổi vị trí của các hàng thành cột và ngược lại trong array.
Trong Pytorch, chuyển vị của tensor có thể được thực hiện bằng cách sử dụng toán tử chuyển vị **.T**, hàm **torch.t()** hoặc **torch.transpose()**. Chuyển vị thay đổi vị trí của các hàng thành cột và ngược lại trong tensor. Cú pháp:

```python
#cách 1:
torch.t(input)

#Cách 2
torch.transpose(input, dim0, dim1)
```

Kết quả trả về là một tensor mới là kết quả của phép chuyển vị. Trong đó:

- input: Tensor cần được chuyển vị.
- dim0, dim1: Các chiều được chọn để thực hiện chuyển vị.

Ví dụ:

```python

#Pytorch code
import torch

#Tạo tensor 2d
tensor_1 = torch.tensor([[1, 2], [3, 4]])
# Chuyển vị tensor
# Cách 1: Sử dụng hàm torch.t()
tensor_transposed_1 = torch.t(tensor_1)
# Cách 2: Sử dụng hàm torch.transpose()
tensor_transposed_2 = torch.transpose(
    tensor_1, 0, 1)
# In ra màn hình
print("tensor 1:\n", tensor_1)
print("tensor 1 sau khi chuyển vị, cách
    1:\n", tensor_transposed_1)
print("tensor 1 sau khi chuyển vị, cách
    2:\n", tensor_transposed_2)
```

```
================ Output ===============
tensor 1:
 tensor([[1, 2],
        [3, 4]])

tensor 1 sau khi chuyển vị, cách 1:
 tensor([[1, 3],
        [2, 4]])

tensor 1 sau khi chuyển vị, cách 2:
 tensor([[1, 3],
        [2, 4]])



=======================================
```

Trong Tensorflow, không có toán tử chuyển vị **T**, mà chỉ dùng hàm **tf.transpose()**. Cách sử dụng cũng tương tự như Numpy và Pytorch:
Ví dụ:

```python
# Tensorflow code
import tensorflow as tf

# Tạo tensor 2d
tensor_1 = tf.constant([[1, 2], [3, 4]])
# Chuyển vị tensor
tensor_transposed = tf.transpose(tensor_1)

# In ra màn hình
print("tensor 1:\n", tensor_1)
print("tensor 1 sau khi chuyển vị:\n",
    tensor_transposed)
```

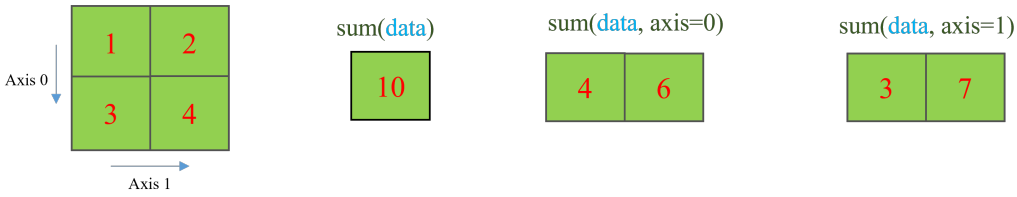
```
================ Output ===============
tensor 1:
 tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tensor 1 sau khi chuyển vị:
 tf.Tensor(
[[1 3]
 [2 4]], shape=(2, 2), dtype=int32)

=======================================
```

b) ***Summation***
Trong thư viện Numpy, hàm **np.sum()** được sử dụng để tính tổng của các phần tử trong mảng (array). Hàm này có thể được áp dụng trên các mảng 1D, 2D hoặc có số chiều cao hơn. Cú pháp:

```python
np.sum(a, axis=None, dtype=None, keepdims=False, initial=0, where=True)
```

Trong đó:

- a: Mảng đầu vào.
- axis: (Tùy chọn) Chiều hoặc các chiều trên đó tổng sẽ được thực hiện. Mặc định là None, tức là tổng của tất cả các phần tử trong mảng.
- dtype: (Tùy chọn) Kiểu dữ liệu của kết quả.
- keepdims: (Tùy chọn) Nếu là True, giữ chiều của mảng kết quả (nếu có) giống với chiều của mảng đầu vào.
- initial: (Tùy chọn) Giá trị khởi tạo cho tổng.
- where: (Tùy chọn) Một mảng Boolean chỉ định vị trí các phần tử được sử dụng trong phép toán.



Hình 2: Minh họa sumation

Trong ví dụ sau, **np.sum()** được sử dụng để tính tổng của mảng array. Tham số sử dụng ở đây là mặc định khi tính tổng các phần tử trong toàn bộ array, sử dụng tham số axis để tính tổng theo cột hoặc hàng. Kết quả được in ra màn hình bao gồm tổng của tất cả các phần tử, tổng theo cột, và tổng theo hàng của mảng.

```python
# Numpy code
import numpy as np

# Tạo mảng 2D
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

# Tính tổng của tất cả các phần tử trong mảng
total_sum = np.sum(arr)

# Tính tổng theo cột (theo chiều dọc)
column_sum = np.sum(arr, axis=0)

# Tính tổng theo hàng (theo chiều ngang)
row_sum = np.sum(arr, axis=1)

# In ra màn hình
print("Mảng:\n", arr)
print("Tổng của tất cả các phần tử trong mảng:\n", total_sum)
print("Tổng theo cột:\n", column_sum)
print("Tổng theo hàng:\n", row_sum)
```

```
================ Output ===============
Mảng:
 [[1 2 3]
 [4 5 6]]

Tổng của tất cả các phần tử trong mảng:
 21

Tổng theo cột:
 [5 7 9]

Tổng theo hàng:
 [ 6 15]

=======================================
```

Trong Pytorch, chúng ta tính sum tương tự như trong Numpy, ví dụ:

```python
#Pytorch code
import torch

# Tạo tensor 2d
tensor_1 = torch.tensor([[1, 2], [3, 4]])
# Tính tổng tensor
tensor_sum = torch.sum(tensor_1)
# Tính tổng tensor theo cột axis = 0
tensor_sum_axis_0 = torch.sum(tensor_1, axis=0)
# Tính tổng tensor theo hàng axis = 1
tensor_sum_axis_1 = torch.sum(tensor_1, axis=1)

# In ra màn hình
print("tensor:\n", tensor_1)
print("Tổng các phần tử trong tensor:\n", tensor_sum)
print("Tổng theo cột:\n", tensor_sum_axis_0)
print("Tổng theo hàng:\n", tensor_sum_axis_1)
```

```
================ Output ===============
tensor:
 tensor([[1, 2],
         [3, 4]])

Tổng các phần tử trong tensor:
 tensor(10)

Tổng theo cột:
 tensor([4, 6])

Tổng theo hàng:
 tensor([3, 7])

=======================================
```

Ví dụ tính sum trong Tensorflow:

```python
# Tensorflow code
import tensorflow as tf

# Tạo tensor 2d
tensor_1 = tf.constant([[1, 2], [3, 4]])
# Tính tổng tensor
tensor_sum = tf.reduce_sum(tensor_1)
# Tính tổng tensor theo cột axis = 0
tensor_sum_axis_0 = tf.reduce_sum(tensor_1
    , axis=0)
# Tính tổng tensor theo hàng axis = 1
tensor_sum_axis_1 = tf.reduce_sum(tensor_1
    , axis=1)

# In ra màn hình
print("tensor:\n", tensor_1)
print("Tổng các phần tử trong tensor:\n",
    tensor_sum)
print("Tổng theo cột:\n",
    tensor_sum_axis_0)
print("Tổng theo hàng:\n",
    tensor_sum_axis_1)
```

```
================ Output ===============
tensor:
 tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

Tổng các phần tử trong tensor:
 tf.Tensor(10, shape=(), dtype=int32)

Tổng theo cột:
 tf.Tensor([4 6], shape=(2,), dtype=int32)

Tổng theo hàng:
 tf.Tensor([3 7], shape=(2,), dtype=int32)




=========================================
```

2. Bài tập

Câu 1: Viết chương trình tạo hai Numpy array, Pytorch tensor, Tensorflow tensor với các giá trị số nguyên ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 4). Sau đó chuyển vị array, tensor thứ 2 và thực hiện phép nhân matrix multiplication. Lưu ý: sử dụng seed=2024

Câu 2: Viết chương trình tạo một Numpy array, Pytorch tensor, Tensorflow tensor với các giá trị số nguyên ngẫu nhiên trong khoảng [-10, 10) với kích thước (3, 3). Sau đó hãy tính tổng của toàn bộ tensor, array, tiếp theo tính tổng theo chiều dọc, chiều ngang . Lưu ý: sử dụng seed=2024

3. Đáp án

Đáp án sẽ được gửi cho các bạn vào khoảng 8h tối trên group Code.