自然語言處理與應用
Natural Language Processing and Applications

PyTorch Tutorial

Instructor: 林英嘉 (Ying-Jia Lin)
2025/03/10

Course GitHub

Slido # NLP_RNN

Preliminary (Python Installation)

- Windows
  - https://itenhanceskills.medium.com/windows-11-安裝-python-44b64a06fd3d
- Mac
  - https://docs.python.org/zh-tw/3.13/using/mac.html
- Or you can use Colab

What is PyTorch?

- PyTorch 是一個開源的機器學習框架，能夠幫助加速從研究原型到商業應用的轉換過程

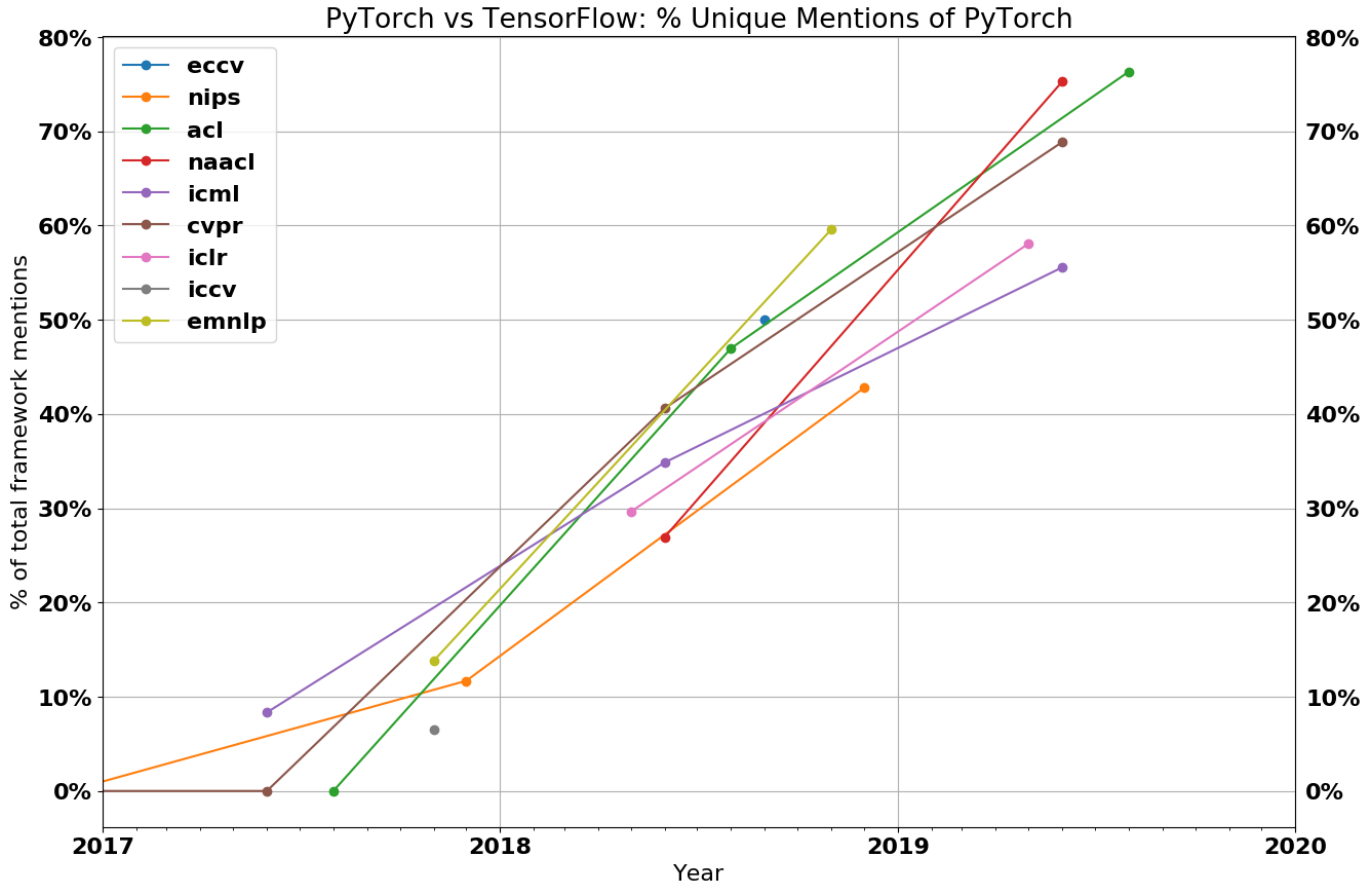
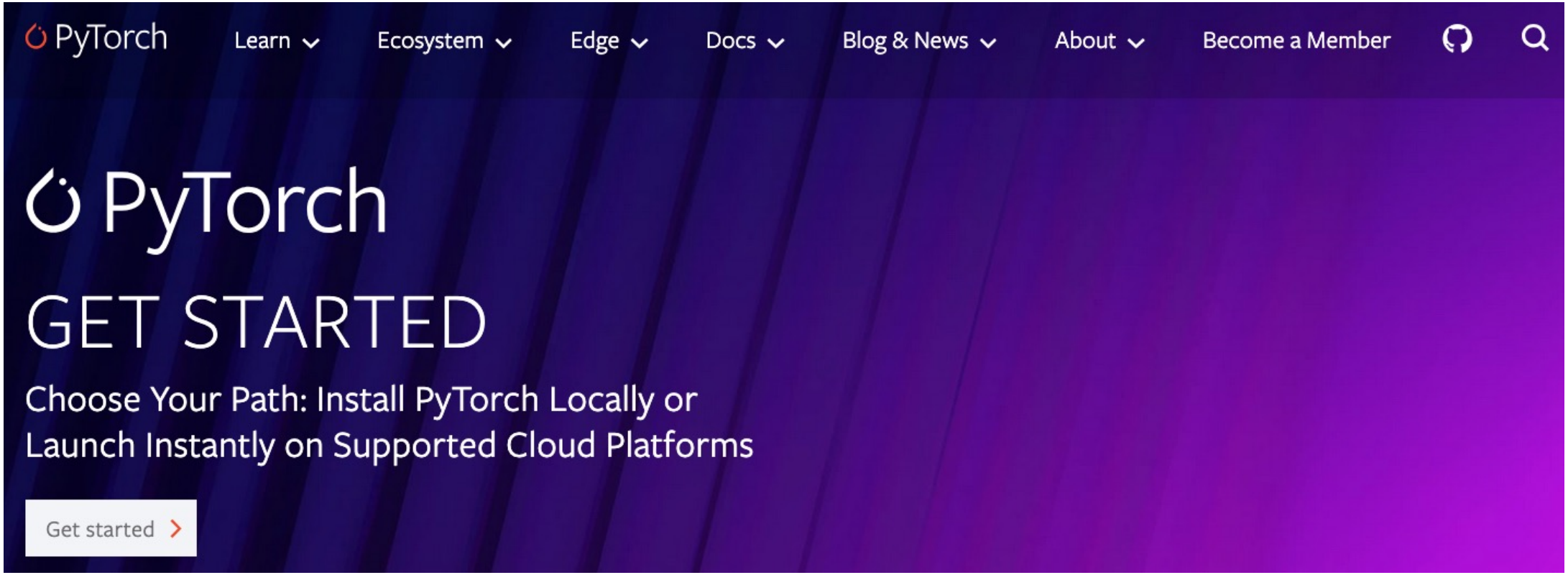


Figure source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

Get Started

- https://pytorch.org



安装 PyTorch

START LOCALLY

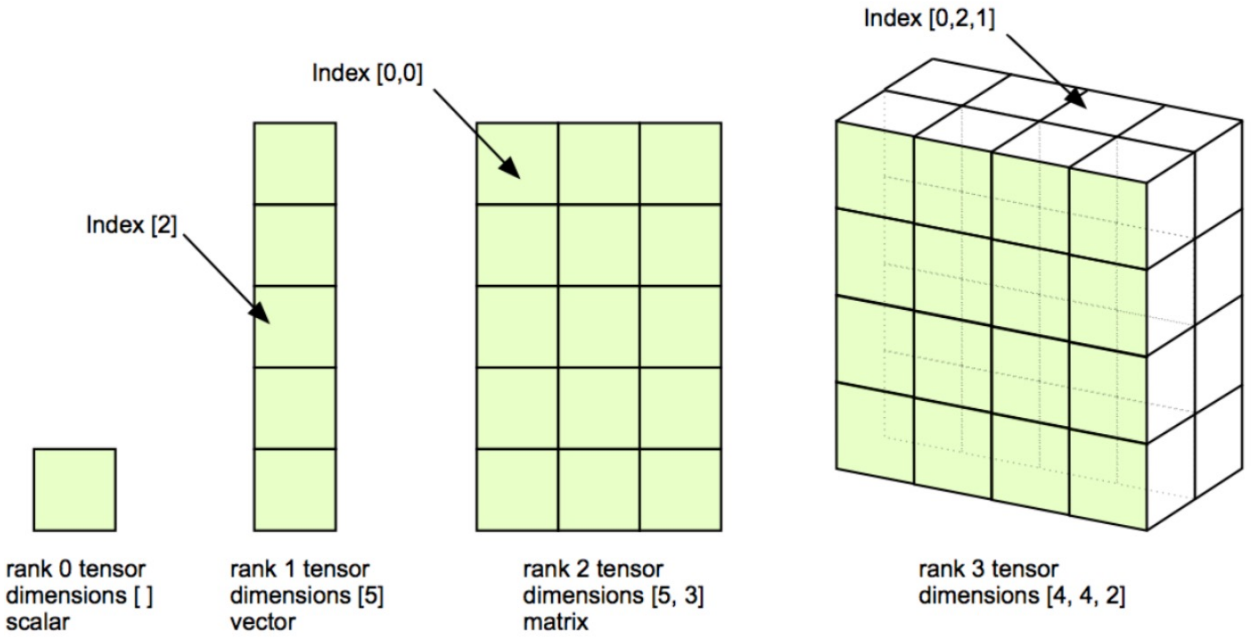
Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

NOTE: Latest PyTorch requires Python 3.9 or later.

PyTorch Build	Stable (2.6.0)			Preview (Nightly)	
Your OS	Linux		Mac		Windows
Package	Conda	Pip		LibTorch	Source
Language	Python			C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.4	CUDA 12.6	ROCm 6.2.4	CPU
Run this Command:	`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`				

The building block in PyTorch

- Tensors (張量) are a specialized data structure that are very similar to arrays and matrices.
- Tensors can be run on CPUs or GPUs (or other hardware accelerators).



https://medium.com/@sahin.samia/understanding-pytorch-basics-through-building-a-logistic-regression-model-from-scratch-71be33a43a00

Initialize a Tensor

*We need to first `import torch`

There are many ways to initialize a PyTorch Tensor:

- From a Python list:

```
t1 = torch.tensor([1, 2, 3])
```

- From NumPy:

```
t8 = torch.tensor(np.array([1., 2., 3.])

arr2 = t8.numpy() # tensor to numpy
```

Initialize a Tensor

There are many ways to initialize a PyTorch Tensor:

- Create an empty tensor:

```
t = torch.empty((2, 3))
```

```
tensor([[0, 0, 0],
        [0, 0, 0]])
```

- Create a tensor with 1 for all elements:

```
t = torch.ones((3, 4))
```

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

- Create a tensor with a value for all elements:

```
t = torch.full((5, 6), 420)
```

```
tensor([[420, 420, 420, 420, 420, 420],
        [420, 420, 420, 420, 420, 420],
        [420, 420, 420, 420, 420, 420],
        [420, 420, 420, 420, 420, 420],
        [420, 420, 420, 420, 420, 420]])
```

Initialize a Tensor-like Tensor

```
t5 = torch.tensor([
    [1, 2, 3],
    [4, 5, 6],
])
```

- Create an empty tensor using the **shape** as t5:

```
t = torch.zeros_like(t5)
```

- Create a tensor with 1 using the **shape** as t5:

```
t = torch.ones_like(t5)
```

- Create a tensor with a value using the **shape** as t5:

```
t = torch.full_like(t5, 420)
```

```
tensor([[0, 0, 0],
        [0, 0, 0]])
tensor([[1, 1, 1],
        [1, 1, 1]])
tensor([[420, 420, 420],
        [420, 420, 420]])
```

以指定數值創造 Tensor

```
# 創造 3x3 單位矩陣
print(torch.eye(3))
print()

# 從 0 列舉至 10，但不包含 10
print(torch.arange(10))
print()

# 從 6 列舉至 9，但不包含 9
print(torch.arange(6, 9))
print()

# 從 4 遞增至 20，但不包含 20，每次遞增 7
print(torch.arange(4, 20, 7))
```

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

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

tensor([6, 7, 8])

tensor([ 4, 11, 18])
```

張量取值 (1)

與 `numpy` 語法概念相似

- 使用 `torch.Tensor[位置]` 來取得 `torch.Tensor` 中指定位置的值
- 若為**多個維度**的張量，則使用 `tuple` 來取得指定位置的值
- 若位置為**負數**，則等同於反向取得指定位置的值
- 取出的值會以 `torch.Tensor.dtype` 的形式保留
- 使用 `torch.Tensor[起始位置:結束位置]` 來取得 `torch.Tensor` 中的部分**連續**值，**包含起始位置**的值，但**不包含結束位置**的值
- 取出的值會以 `torch.Tensor` 的形式保留

張量取值 (1)

- 取出的值會以 `torch.Tensor` 的形式保留

- 使用 `torch.Tensor[位置]` 來取得 `torch.Tensor` 中指定位置的值

```
# 宣告 Tensor 變數
t9 = torch.tensor([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11],
])

# 輸出張量 t9 中的第 0 個位置的值 [0, 1, 2]
print(t9[0])
# 輸出張量 t9 中的第 1 個位置的值 [3, 4, 5]
print(t9[1])
# 輸出張量 t9 中的第 -2 個位置的值 [6, 7, 8]
print(t9[-2])
# 輸出張量 t9 中的第 -1 個位置的值 [9, 10, 11]
print(t9[-1])
```

張量取值 (2)

- 取出的值會以 `torch.Tensor` 的形式保留

- 若為**多個維度**的張量，則可輸入多個位置來取得指定位置的值

```
# 宣告 Tensor 變數
t9 = torch.tensor([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11],
])

# 輸出張量 t9 中的第 [0, 1] 個位置的值 1
print(t9[0, 1])
# 輸出張量 t9 中的第 [1, 2] 個位置的值 5
print(t9[1, 2])
# 輸出張量 t9 中的第 [-1, -1] 個位置的值 11
print(t9[-1, -1])
# 輸出張量 t9 中的第 [-2, -1] 個位置的值 8
print(t9[-2, -1])
```

張量取值 (3)

- 取出的值會以 `torch.Tensor` 的形式保留

- 使用 `torch.Tensor[起始位置:結束位置]` 來取得 `torch.Tensor` 中的部分**連續**值

(包含起始位置的值，但**不包含結束位置**的值)

```
# 宣告 Tensor 變數
t10 = torch.tensor([
0, 10, 20, 30, 40,
50, 60, 70, 80, 90
])

# 輸出張量 t10 位置 0, 1, 2 但是不含位置 3 的值 [0, 10, 20]
print(t10[0:3])
# 輸出張量 t10 位置 7, 8, 9 的值 [70, 80, 90]
print(t10[7:])
# 輸出張量 t10 位置 0, 1 但是不含位置 2 的值 [0, 10]
print(t10[:2])
# 輸出張量 t10 所有位置的值 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
print(t10[:])
```

張量取值 (4)

- 取出的值會以 `torch.Tensor` 的形式保留

- 使用 `torch.Tensor[起始位置:結束位置]` 來取得 `torch.Tensor` 中的部分**連續**值

(包含起始位置的值，但**不包含結束位置**的值)

```
# 宣告 Tensor 變數
t11 = torch.tensor([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11],
])
# 輸出張量 t11 位置 0, 1, 但是不含位置 2 的值 [[0, 1, 2], [3, 4, 5]]
print(t11[0:2])
# 輸出張量 t11 位置 0 但是不含位置 1 的值 [[0, 1, 2]]
print(t11[:1])
# 輸出張量 t11 全部的值 [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
print(t11[:])
```

張量取值 (5)

- 取出的值會以 `torch.Tensor` 的形式保留

- 使用 `torch.Tensor[iterable]`（例如 `list`, `tuple` 等）來取得**多個** `torch.Tensor` 中的值

```
# 宣告 Tensor 變數
t13 = torch.tensor([
[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]
])

# 輸出張量 t13[0] 與 t13[1] 的值 [[1, 2, 3, 4] [5, 6, 7, 8]]
print(t13[[0, 1]])
# 輸出張量 t13[0, 2] 與 t13[1, 3] 的值 [3, 8]
print(t13[[0, 1], [2, 3]])
```

張量取值 (6)

- 取出的值會以 `torch.Tensor` 的形式保留

- 使用判斷式來取得 `torch.Tensor` 中的部份資料
  - 經由判斷式所得結果也為 `torch.Tensor`
  - 判斷式所得結果之 `torch.Tensor.dtype` 為**布林值** `bool`（`True` 或 `False`）

```
# 宣告 Tensor 變數
t15 = torch.tensor([
0, 10, 20, 30, 40,
50, 60, 70, 80, 90
])

# 輸出每個值是否大於 50 的 `torch.Tensor`
print(t15 > 50)
# 輸出 torch.bool
print((t15 > 50).dtype)
# 輸出大於 50 的值 [60, 70, 80, 90]
print(t15[t15 > 50])
# 輸出除以 20 餘數為 0 的值 [0, 20, 40, 60, 80]
print(t15[t15 % 20 == 0])
```

純量運算 (Scalar Operation)

- 對張量內所有數值與單一純量（`Scalar`）進行相同計算。

符號	意義
`torch.Tensor + scalar`	張量中的每個數值加上 `scalar`
`torch.Tensor - scalar`	張量中的每個數值減去 `scalar`
`torch.Tensor * scalar`	張量中的每個數值乘上 `scalar`
`torch.Tensor / scalar`	張量中的每個數值除以 `scalar`
`torch.Tensor // scalar`	張量中的每個數值除以 `scalar` 所得之商
`torch.Tensor % scalar`	張量中的每個數值除以 `scalar` 所得之餘數
`torch.Tensor ** scalar`	張量中的每個數值取 `scalar` 次方

個別數值運算 (Element-wise Operation)

- 若兩個張量想要進行運算，則兩個張量的維度必須相同（即兩張量之 `torch.size()` 相同）。

符號	意義
`A + B`	張量 `A` 中的每個數值加上張量 `B` 中相同位置的數值
`A - B`	張量 `A` 中的每個數值減去張量 `B` 中相同位置的數值
`A * B`	張量 `A` 中的每個數值乘上張量 `B` 中相同位置的數值
`A / B`	張量 `A` 中的每個數值除以張量 `B` 中相同位置的數值
`A // B`	張量 `A` 中的每個數值除以張量 `B` 中相同位置的數值所得之商
`A % B`	張量 `A` 中的每個數值除以張量 `B` 中相同位置的數值所得之餘數
`A ** B`	張量 `A` 中的每個數值取張量 `B` 中相同位置的數值之次方

個別數值函數運算 (Element-wise Functional Operation)

- 我們可以透過 PyTorch 提供的函數進行張量運算。

函數	意義
`torch.sin`	張量中的每個數值 x 計算 $\sin(x)$
`torch.cos`	張量中的每個數值 x 計算 ...
`torch.tan`	張量中的每個數值 x 計算 $\tan(x)$
`torch.exp`	張量中的每個數值 x 計算 e^x
`torch.log`	張量中的每個數值 x 計算 $\log x$
`torch.ceil`	張量中的每個數值 x 計算 $\lceil x \rceil$
`torch.floor`	張量中的每個數值 x 計算 $\lfloor x \rfloor$

張量自動擴充 (Broadcasting)

若張量 `A` 的維度為 `(a1, a2, ..., an)`（即 `A.size() == (a1, a2, ..., an)`），則張量 `B` 在滿足以下其中一種條件時即可與張量 `A` 進行運算：

1. 張量 `B` 與張量 `A` 維度完全相同（即 `B.size() == (a1, a2, ..., an)`）
2. 張量 `B` 為純量（即 `B.size() == (1,)`）
3. 張量 `B` 的維度為 `(b1, b2, ..., bn)`，若 `ai != bi`，則 `ai == 1` 或 `bi == 1`
    - 從**最後**一個維度開始比較
    - 如果有任何一個維度無法滿足前述需求，則會得到 `ValueError`

Broadcasting Example

```
t19 = torch.tensor([
    [
        [1, 2],
        [3, 4],
        [5, 6],
    ],
    [
        [7, 8],
        [9 ,10],
        [11, 12]
    ]
])
```

shape: 2x3x2

```
t20 = torch.tensor([
    [
        [1],
        [1],
        [1]
    ],
    [
        [2],
        [2],
        [2]
    ],
])
```

shape: 2x3x1

```
# t19+ t19
tensor([[[ 2,  4],
         [ 6,  8],
         [10, 12]],

        [[14, 16],
         [18, 20],
         [22, 24]]])
# t19+ t20
tensor([[[ 2,  3],
         [ 4,  5],
         [ 6,  7]],

        [[ 9, 10],
         [11, 12],
         [13, 14]]])
```

高維張量運算

- 矩陣等同於是維度為 2 的張量。而**高維度**的張量運算等同於**固定大部分的維度**，只使用**其中的兩個維度進行計算**

張量乘法（Tensor Multiplication）

例如：以 $A.\text{size}() = (4, 3)$ 與 $B.\text{size}() = (3, 2)$ 來說，$(A \times B).\text{size}() = (4, 2)$。

例如：以 $A.\text{size}() = (5, 4, 3)$ 與 $B.\text{size}() = (5, 3, 2)$ 來說，$(A \times B).\text{size}() = (5, 4, 2)$。

例如：以 $A.\text{size}() = (1995, 10, 12, 5, 4, 3)$ 與 $B.\text{size}() = (1995, 10, 12, 5, 3, 2)$ 來說，$(A \times B).\text{size}() = (1995, 10, 12, 5, 4, 2)$。

高維張量運算範例

- 矩陣等同於是維度為 2 的張量。而**高維度**的張量運算等同於**固定大部分的維度**，只使用**其中的兩個維度進行計算**

```
# 宣告 Tensor 變數
t22 = torch.ones(5, 4, 3)
# 宣告 Tensor 變數
t23 = torch.ones(5, 3, 2)
# 進行張量乘法
t24 = torch.matmul(t22, t23)

# 輸出張量 t22 的維度
print(t22.size())
# 輸出張量 t23 的維度
print(t23.size())
# 輸出張量 t24 的維度
print(t24.size())
```

```
torch.Size([5, 4, 3])
torch.Size([5, 3, 2])
torch.Size([5, 4, 2])
```

張量轉置

```
# 宣告 Tensor 變數
t28 = torch.ones(5, 4, 3)

# 輸出轉置維度 1 與 2 後的維度
print(torch.transpose(t28, 1, 2).size())
# 輸出轉置維度 1 與 2 後的維度
print(t28.transpose(1, 2).size())

# 輸出轉置維度 0 與 2 後的維度
print(torch.transpose(t28, 0, 2).size())
# 輸出轉置維度 0 與 2 後的維度
print(t28.transpose(0, 2).size())
```

```
torch.Size([5, 3, 4])
torch.Size([5, 3, 4])
torch.Size([3, 4, 5])
torch.Size([3, 4, 5])
```

.view() and .reshape()

view 與 reshape

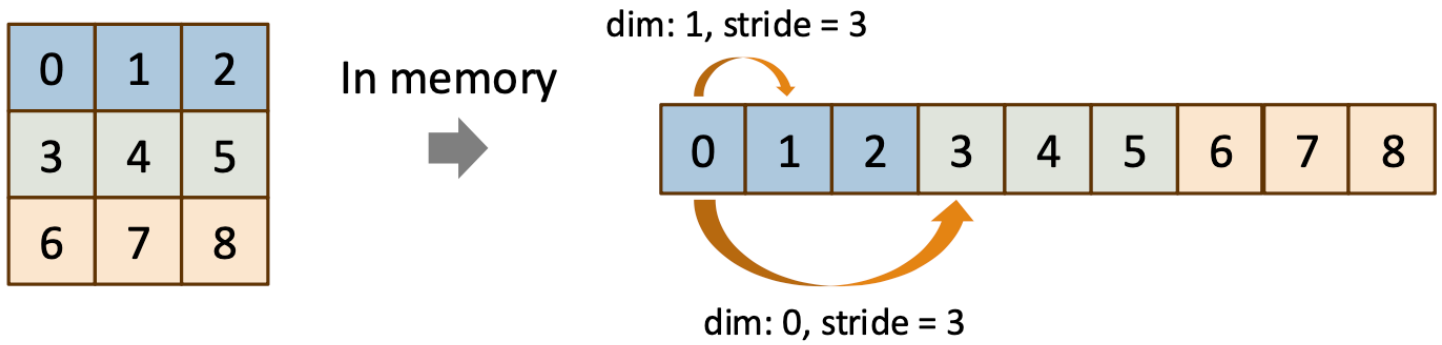
`view()` 與 `reshape()` 都可以用來改變 Tensor 的形狀，但它們有一些關鍵的區別：

- 記憶體連續性（Memory Contiguity）
  - `view()` 要求原 Tensor 在記憶體中是連續的（contiguous），否則會報錯。
  - `reshape()` 不強制要求記憶體連續，如果 Tensor 不是 contiguous 的，它會自動調用 .contiguous() 後再重新排列數據。
- 內部實現
  - `view()` 不會改變資料本身，只是改變視圖（view），它返回的是與原 Tensor 共享資料的 Tensor。
  - `reshape()` 可能會返回原資料的視圖，也可能會創建新的 Tensor 並複製資料，具體取決於記憶體分配。

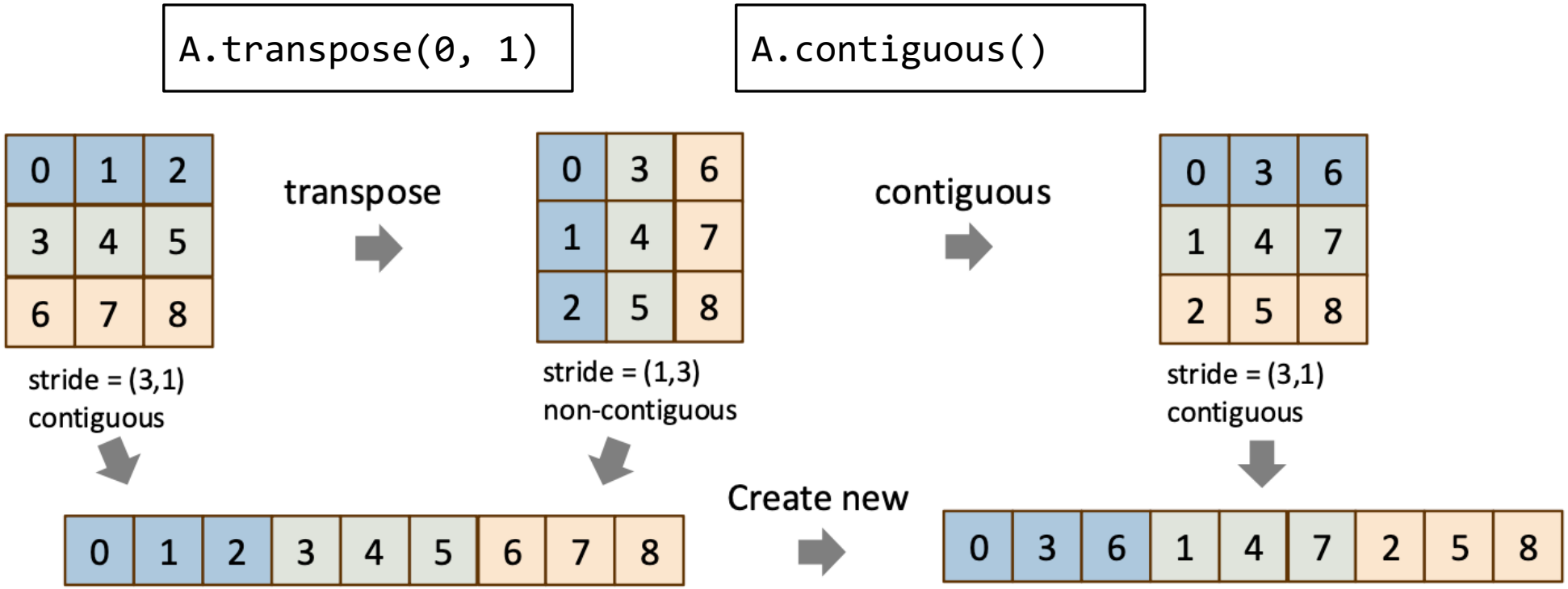
.stride()

- `stride` 代表沿著某個維度前進 `1` 步時，對應的元素在記憶體中需要跳多少個位置
- `contiguous()` 需要符合 **row-major order**

```
A = torch.arange(9).reshape(3, 3)
A.stride() # (3, 1)
```



.stride() and .transpose()



降維函數 (Dimension Decreasing Function)

- 以下函數將會使輸出張量維度小於輸入張量維度

函數	意義
torch.sum	將所有數值相加
torch.max	取出所有數值中最大者
torch.min	取出所有數值中最小者
torch.argmax	取出所有數值中最大者的位置
torch.argmin	取出所有數值中最小者的位置
torch.mean	取出所有數值的平均值
torch.var	取出所有數值的變異數
torch.std	取出所有數值的標準差
torch.squeeze	移除數字為 1 的維度

.argmax()

```
t31 = torch.tensor([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])
print(torch.argmax(t31)) # tensor(11)
print(torch.argmax(t31, dim=0)) # tensor([2, 2, 2, 2])
print(torch.argmax(t31, dim=1)) # tensor([3, 3, 3])
```

torch.squeeze()

```
# 宣告 Tensor 變數
t33 = torch.tensor([[1, 2, 3]])

# 移除張量 t33 中多餘的維度(為 1 的維度)
t33_sq = torch.squeeze(t33)

# 輸出張量 t33
print(t33)
# 輸出張量 t33 的維度
print('- before squeezing:', t33.size())
# 輸出移除維度後張量 t33 的維度
print('- after squeezing:',t33_sq.size())
```

```
tensor([[1, 2, 3]])
- before squeezing: torch.Size([1, 3])
- after squeezing: torch.Size([3])
```

torch.cat()

- cat: concatenation

```
# 增維函數 (torch.cat)
t34 = torch.tensor([[1, 2, 3]])

# 串接多個張量 t34
t34_cat = torch.cat([
    t34,
    t34,
    t34,
    t34
])

# 輸出串接後的張量 t34_cat
print(t34_cat)
# 輸出串接後的張量 t34_cat 維度
print(t34_cat.size())
```

```
tensor([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]])
torch.Size([4, 3])
```

torch.unsqueeze()

```
# 增維函數 (torch.unsqueeze)

# 宣告 Tensor 變數
t35 = torch.tensor([
    [1, 2, 3],
    [4, 5, 6]
])

print(t35.size())

# 對張量 t35 維度 0 增加 1 維
t35_usq = torch.unsqueeze(t35, dim=0)

# 輸出張量 t35 維度 0 增加 1 維後的結果
print(t35_usq)
# 輸出張量 t35 維度 0 增加 1 維後的維度
print(t35_usq.size())
```

```
torch.Size([2, 3])
tensor([[[1, 2, 3],
         [4, 5, 6]]])
torch.Size([1, 2, 3])
```

Thank you!

Instructor: 林英嘉

✉ yjlin@cgu.edu.tw

TA: 吳宣毅

✉ m1161007@cgu.edu.tw