2. PyTorch Basic

1. 파이토치 개요
2. 파이토치 환경설정
3. 파이토치 기초문법

1. 파이토치 개요

❖ 파이토치 개요

- 파이토치(PyTorch)는 2017년 초에 공개된 딥러닝 프레임워크로 루아(Lua) 언어로 개발되었던 토치(Torch)를 페이스북에서 파이썬 버전으로 내놓은 것
- 토치는 파이썬의 넘파이(NumPy) 라이브러리처럼 과학 연산을 위한 라이브러리로 공개되었지만 이후 발전을 거듭하면서 딥러닝 프레임워크로 발전
- 파이토치 공식 튜토리얼에서는 파이토치를 다음과 같이 언급하고 있음
- 파이썬 기반의 과학 연산 패키지로 다음 두 집단을 대상으로 함
  - 넘파이를 대체하면서 GPU를 이용한 연산이 필요한 경우
  - 최대한의 유연성과 속도를 제공하는 딥러닝 연구 플랫폼이 필요한 경우
- 무엇보다 주목받는 이유 중 하나는 간결하고 빠른 구현성에 있음

1. 파이토치 개요

❖ 파이토치 특징 및 장점

- 파이토치 특징은 다음과 같이 한마디로 특징 지을 수 있음

  GPU에서 텐서 조작 및 동적 신경망 구축이 가능한 프레임워크

- GPU, 텐서, 동적 신경망이란 무엇을 의미할까?
  - GPU(Graphics Processing Unit): 연산 속도를 빠르게 하는 역할
  - 딥러닝에서는 기울기를 계산할 때 미분을 쓰는데, GPU를 사용하면 빠른 계산이 가능
  - 내부적으로 CUDA, cuDNN이라는 API를 통해 GPU를 연산에 사용할 수 있음
  - 병렬 연산에서 GPU의 속도는 CPU의 속도보다 훨씬 빠르므로 딥러닝 학습에서 GPU 사용은 필수라고 할 수 있음

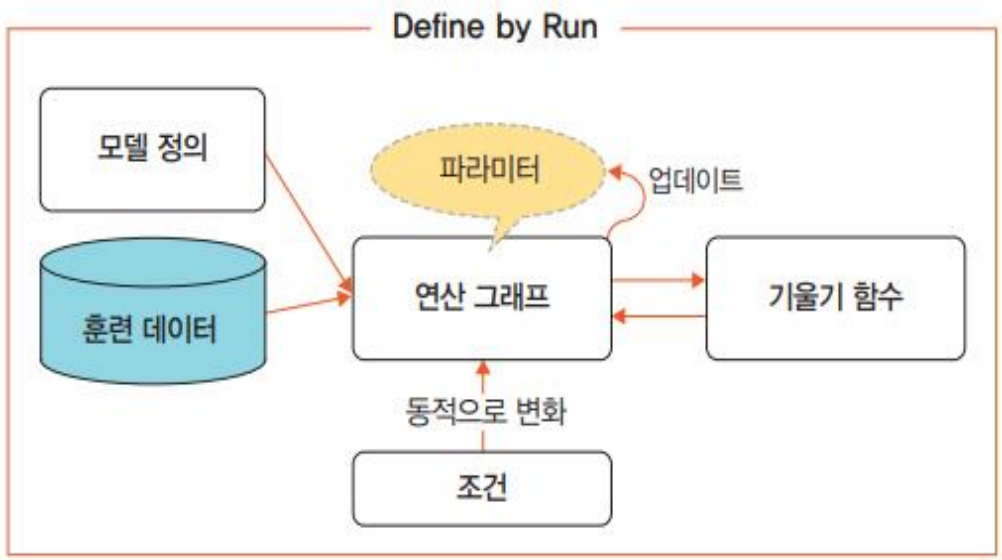
1. 파이토치 개요

❖ 파이토치 특징 및 장점

- **텐서**(Tensor): 텐서는 파이토치의 데이터 형태
  - 텐서는 단일 데이터 형식으로 된 자료들의 다차원 행렬
  - 텐서는 간단한 명령어(변수 뒤에 .cuda()를 추가)를 사용해서 GPU로 연산을 수행하게 할 수 있음

- **동적 신경망**: 훈련을 반복할 때마다 네트워크 변경이 가능한 신경망을 의미
  - 예: 학습 중에 은닉층을 추가하거나 제거하는 등 모델의 네트워크 조작이 가능
  - 연산 그래프를 정의하는 것과 동시에 값도 초기화되는 'Define by Run' 방식을 사용
  - 연산 그래프와 연산을 분리해서 생각할 필요가 없기 때문에 코드를 이해하기 쉬움

▼ 그림 2-1 파이토치 'Define by Run'

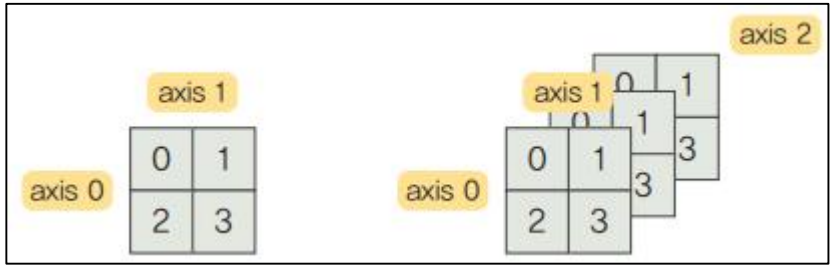


1. 파이토치 개요

❖ 파이토치 특징 및 장점

- 벡터, 행렬, 텐서
  - 인공지능(머신 러닝/딥러닝)에서 데이터는 벡터(vector)로 표현
  - **벡터(vector)** : [1.0, 1.1, 1.2]처럼 숫자들의 리스트로, 1차원 배열 형태
  - **행렬(matrix)** : 행과 열로 표현되는 2차원 배열 형태, 이때 가로줄을 행(row)이라고 하며, 세로줄을 열(column)이라고 함
  - **텐서(tensor)** : 3차원 이상의 배열 형태
    - 1차원 축(행)=axis 0=벡터
    - 2차원 축(열)=axis 1=행렬
    - 3차원 축(채널)=axis 2=텐서

▼ 그림 2-2 벡터, 행렬, 텐서



1. 파이토치 개요

❖ 파이토치 특징 및 장점

- 행렬은 복수의 차원을 가지는 데이터 레코드의 집합
- 이때 하나의 데이터 레코드를 벡터 단독으로 나타낼 때는 다음과 같이 하나의 열로 표기

$$x_1 = \begin{bmatrix} 1.1 \\ 2.7 \\ 3.3 \\ 0.2 \end{bmatrix} \quad x_2 = \begin{bmatrix} 4.5 \\ 1.2 \\ 0.7 \\ 3.5 \end{bmatrix}$$

- 반면에 복수의 데이터 레코드 집합을 행렬로 나타낼 때는 다음과 같이 하나의 데이터 레코드가 하나의 행으로 표기

$$X = \begin{bmatrix} 1.1 & 2.7 & 3.3 & 0.2 \\ 4.5 & 1.2 & 0.7 & 3.5 \end{bmatrix}$$

- 즉, 행렬의 일반적인 표현은 다음과 같음

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

- 텐서는 행렬의 다차원 표현이라고 생각하면 쉬움
- 같은 크기의 행렬이 여러 개 묶여 있는 것으로 다음과 같이 표현할 수 있음

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

1. 파이토치 개요

❖ 파이토치 특징 및 장점

- 파이토치에서 텐서를 표현하기 위해서는 다음 코드와 같이 torch.tensor()를 사용

```
import torch
torch.tensor([[1., -1.], [1., -1.]])
```

- 생성된 텐서의 형태는 다음과 같이 표현

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

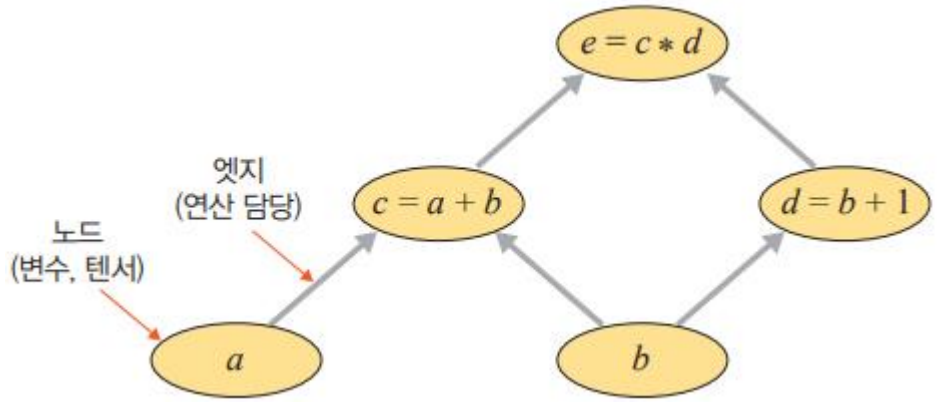
- 벡터, 행렬 등 자세한 내용은 선형대수학 도서를 참고

1. 파이토치 개요

❖ 파이토치 특징 및 장점

- 연산 그래프
  - 연산 그래프는 방향성이 있으며 변수(텐서)를 의미하는 노드와 연산(곱하기, 더하기)을 담당하는 엣지로 구성
  - 다음 그림과 같이 노드는 변수(a, b)를 가지고 있으며 각 계산을 통해 새로운 텐서(c, d, e)를 구성할 수 있음

▼ 그림 2-3 파이토치 연산 그래프



- 신경망은 연산 그래프를 이용하여 계산을 수행
- 즉, 네트워크가 학습될 때 손실 함수의 기울기가 가중치와 바이어스를 기반으로 계산되며, 이후 경사 하강법을 사용하여 가중치가 업데이트
- 이때 연산 그래프를 이용하여 이 과정이 효과적으로 수행

1. 파이토치 개요

❖ 파이토치 특징 및 장점

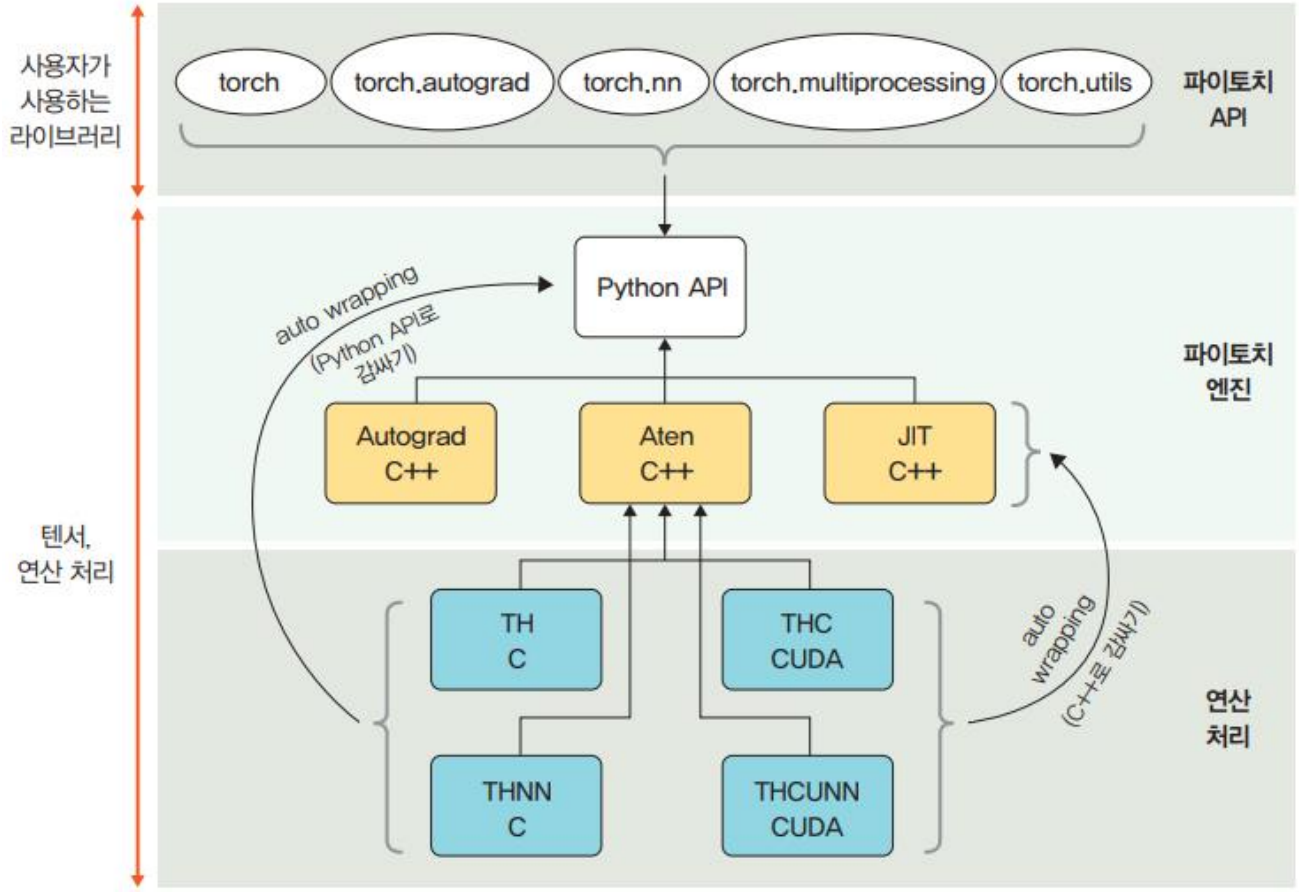
- 파이토치는 효율적인 계산, 낮은 CPU 활용, 직관적인 인터페이스와 낮은 진입 장벽 등을 장점으로 꼽을 수 있음
  - 단순함(효율적인 계산)
  - 파이썬 환경과 쉽게 통합할 수 있음
  - 디버깅이 직관적이고 간결함
- 성능(낮은 CPU 활용)
  - 모델 훈련을 위한 CPU 사용률이 텐서플로와 비교하여 낮음
  - 학습 및 추론 속도가 빠르고 다루기 쉬움
- 직관적인 인터페이스
  - 텐서플로처럼 잦은 API 변경(layers → slim → estimators → tf.keras)이 없어 배우기 쉬움

1. 파이토치 개요

❖ 파이토치의 아키텍처

- 크게 세 개의 계층으로 나누어 설명할 수 있음
- 파이토치 API : 가장 상위 계층은 파이토치 API가 위치해 있으며 사용자 라이브러리
- 파이토치 엔진 : 다차원 텐서 및 자동 미분을 처리
- 연산처리 : 마지막으로 가장 아래에는 텐서에 대한 연산을 처리
- CPU/GPU를 이용하는 텐서의 실질적인 계산을 위한 C, CUDA 등 라이브러리가 위치

▼ 그림 2-4 파이토치의 아키텍처



1. 파이토치 개요

❖ 파이토치의 아키텍처(파이토치 API)

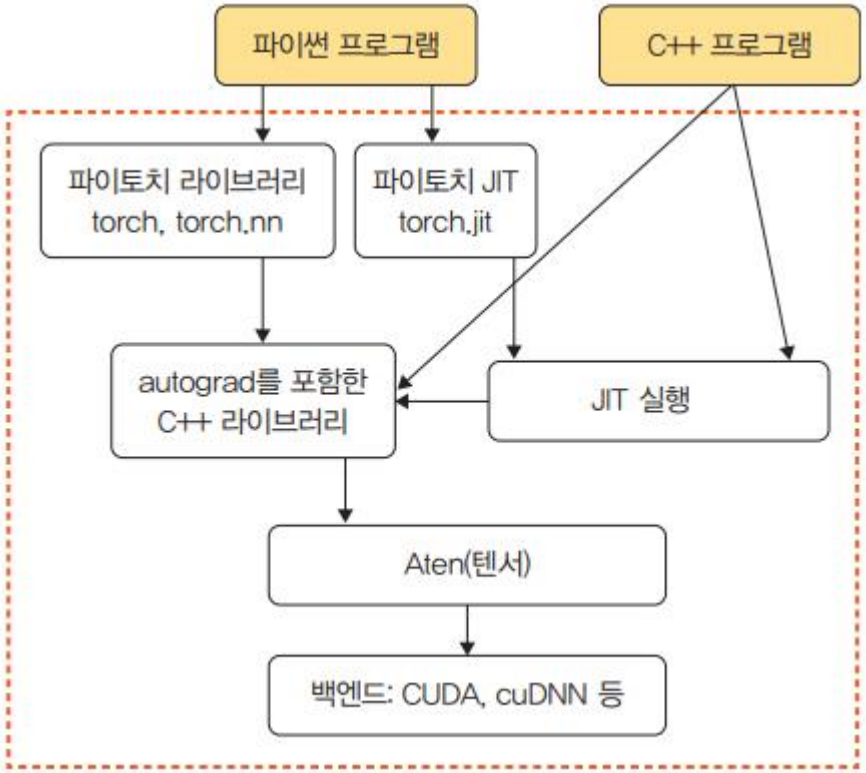
- 사용자가 이해하기 쉬운 API를 제공하여 텐서에 대한 처리와 신경망을 구축하고 훈련할 수 있도록 도움
- 사용자 인터페이스를 제공하지만 실제 계산은 수행하지 않음
- 그 대신 C++로 작성된 파이토치 엔진으로 그 작업을 전달하는 역할만 함
- 파이토치 API 계층에서는 사용자의 편의성을 위해 다음 패키지들이 제공
  - torch: GPU를 지원하는 텐서 패키지
  - torch.autograd: 자동 미분 패키지
  - torch.nn: 신경망 구축 및 훈련 패키지
  - torch.multiprocessing: 파이썬 멀티프로세싱 패키지
  - torch.utils: DataLoader 및 기타 유틸리티를 제공하는 패키지

1. 파이토치 개요

❖ 파이토치의 아키텍처(파이토치 엔진)

- 파이토치 엔진은 Autograd C++, Aten C++, JIT C++, Python API로 구성
- Autograd C++는 가중치, 바이어스를 업데이트하는 과정에서 필요한 미분을 자동으로 계산해 주는 역할
- Aten C++는 C++ 텐서 라이브러리를 제공
- JIT C++는 계산을 최적화하기 위한 JIT(Just In-Time) 컴파일러
- 파이토치 엔진 라이브러리는 C++로 감싼(래핑(wrapping)) 다음 Python API 형태로 제공되기 때문에 사용자들이 손쉽게 모델을 구축하고 텐서를 사용할 수 있음

▼ 그림 2-5 파이토치 엔진



1. 파이토치 개요

❖ 파이토치의 아키텍처(연산 처리)

- 가장 아래 계층에 속하는 C 또는 CUDA 패키지는 상위의 API에서 할당된 거의 모든 계산을 수행
- 이 층에서 제공되는 패키지는 CPU와 GPU(TH(토치), THC(토치 CUDA))를 이용하여 효율적인 데이터 구조, 다차원 텐서에 대한 연산을 처리

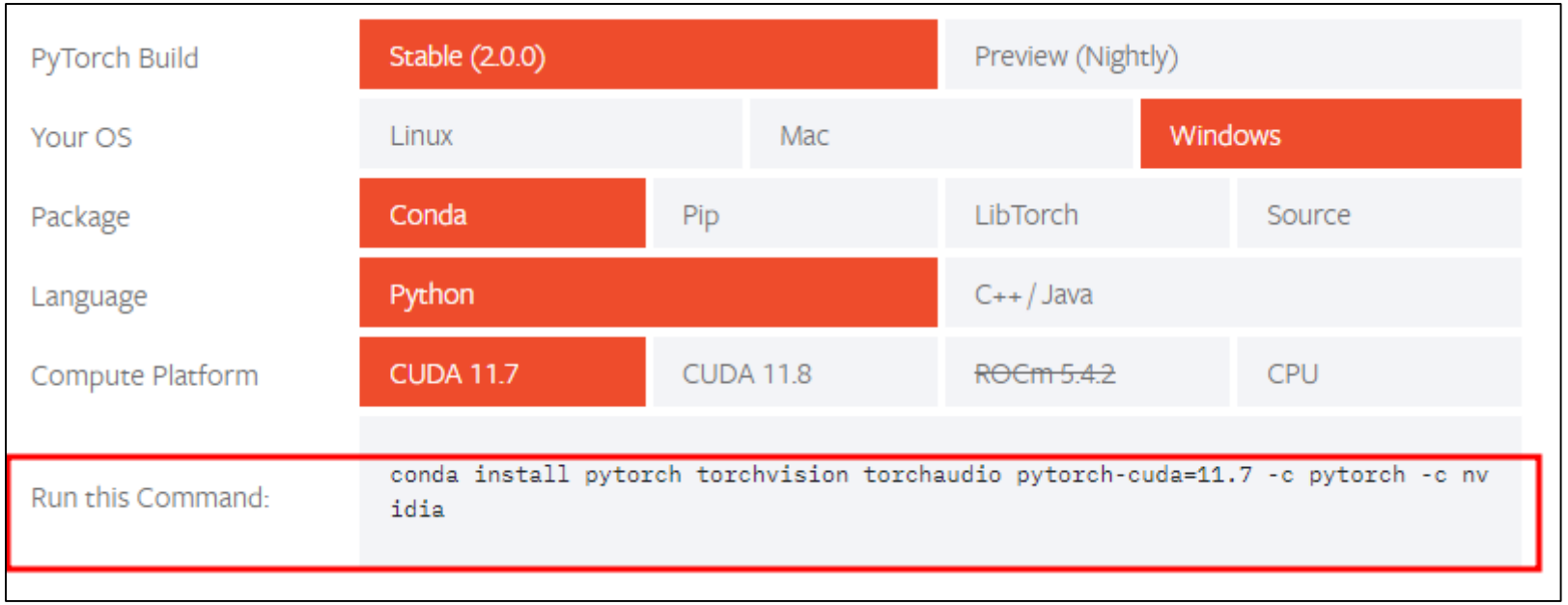
2. 파이토치 환경설정

1. 아나콘다 설치

- https://www.anaconda.com/download/ 웹 사이트에서 자신에게 맞는 버전에 맞는 64-Bit Graphical Installer 내려받음

2. Pytorch 설치

1. https://pytorch.org/get-started/locally/ 사이트에 접속하여 자신에 맞는 운영체제와 CUDA를 선택하고 package 부분에 Conda를 선택 한 후 Run this Command 부분의 설치 명령어를 anaconda Prompt에 복사하여 설치한다.



3. 파이토치 기초 문법

❖ 텐서 다루기– 텐서 생성

- 텐서는 파이토치의 가장 기본이 되는 데이터 구조
- 넘파이의 ndarray와 비슷하며 GPU에서의 연산도 가능
- 텐서 생성은 다음과 같은 코드를 이용

```
# type별 tensor 생성
ft = torch.FloatTensor([[1, 2],[3, 4]]); ft
lt = torch.LongTensor([[1, 2],[3, 4]]); lt
bt = torch.ByteTensor([[1, 0],[0, 1]]); bt
```

```
import torch
print(torch.tensor([[1,2],[3,4]])) ------ 2차원 형태의 텐서 생성
print(torch.tensor([[1,2],[3,4]], device="cuda:0")) ------ GPU에 텐서 생성
print(torch.tensor([[1,2],[3,4]], dtype=torch.float64)) ------ dtype을 이용하여 텐서 생성
```

```
tensor([[1, 2],
        [3, 4]])

tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

◀결과

3. 파이토치 기초 문법

❖ 텐서 다루기– 넘파이 호환

- ndarray–〉 torch tensor 변환 –〉 ndarray

```
import numpy as np

# Define numpy array.
x = np.array([[1, 2],[3, 4]])
print(x, type(x))
```

```
[[1 2]
 [3 4]] <class 'numpy.ndarray'>
```

```
x = torch.from_numpy(x)
print(x, type(x))
```

```
tensor([[1, 2],
        [3, 4]]) <class 'torch.Tensor'>
```

```
x = x.numpy()
print(x, type(x))
```

```
[[1 2]
 [3 4]] <class 'numpy.ndarray'>
```

3. 파이토치 기초 문법

❖ 텐서 다루기– 넘파이 호환

- 텐서 –〉 ndarray로 변환

```
temp = torch.tensor([[1,2],[3,4]])
print(temp.numpy()) ------ 텐서를 ndarray로 변환


temp = torch.tensor([[1,2],[3,4]], device="cuda:0")
print(temp.to("cpu").numpy()) ------ GPU상의 텐서를 CPU의 텐서로 변환한 후 ndarray로 변환
```

```
[[1 2]
 [3 4]]

[[1 2]
 [3 4]]
```

◀결과

3. 파이토치 기초 문법

❖ 텐서의 데이터 형(type) 변환

float -> long

```
ft.long()
```

```
tensor([[1, 2],
        [3, 4]])
```

long -> float

```
lt.float()
```

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

float tensor 생성 -> byte

```
torch.FloatTensor([1, 0]).byte()
```

```
tensor([1, 0], dtype=torch.uint8)
```

3. 파이토치 기초 문법

❖ 텐서 크기 구하기

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

```
print(x.size())
print(x.shape)
```

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

```
print(x.dim())
print(len(x.size()))
```

```
3
3
```

```
print(x.size(1))
print(x.shape[1])
```

```
2
2
```

```
print(x.size(-1))
print(x.shape[-1])
```

```
2
2
```

3. 파이토치 기초 문법

❖ 기본 연산(+ - * / == != **)

```
a = torch.FloatTensor([[1, 2],
                       [3, 4]])
b = torch.FloatTensor([[2, 2],
                       [3, 3]])
```

```
a + b
```

```
tensor([[3., 4.],
        [6., 7.]])
```

```
a - b
```

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

```
a * b
```

```
tensor([[ 2.,  4.],
        [ 9., 12.]])
```

```
a / b
```

```
tensor([[0.5000, 1.0000],
        [1.0000, 1.3333]])
```

```
a == b
```

```
tensor([[False,  True],
        [ True, False]])
```

```
a != b
```

```
tensor([[ True, False],
        [False,  True]])
```

```
a ** b
```

```
tensor([[ 1.,  4.],
        [27., 64.]])
```

3. 파이토치 기초 문법

❖ 인플레이스 연산(Inplace Operations)

❖ Sum,Mean(Dimension Reducing Operations)

```
# inplace
print(a)
print(a.mul(b))
print(a)

print(a.mul_(b))
print(a)
```

```
# Dimension reducing
x = torch.FloatTensor([[1, 2], [3, 4]])
print(x.sum())
print(x.mean())
print(x.sum(dim=0))
print(x.sum(dim=-1))
```

3. 파이토치 기초 문법

❖ 브로드캐스트 연산

```
# 텐서+ 스칼라

x = torch.FloatTensor([[1, 2],
                       [3, 4]])
y = 1

print(x.size())

z = x + y
print(z)
print(z.size())
```

```
# 텐서+ 벡터

x = torch.FloatTensor([[1, 2], [4, 8]])
y = torch.FloatTensor([3,  5])

print(x.size())
print(y.size())

z = x + y
print(z)
print(z.size())

x1 = torch.FloatTensor([[[1, 2]]])
y2 = torch.FloatTensor([3,  5])
z1=x1 +y1  =>?
```

```
# 텐서+텐스

x = torch.FloatTensor([[1, 2]])
y = torch.FloatTensor([[3], [5]])

print(x.size())
print(y.size())

z = x + y
print(z)
print(z.size())

x2 = torch.FloatTensor([[[1, 2], [4, 8]]])
y2 = torch.FloatTensor([[1, 2, 3],
                        [4, 5, 6], [7, 8, 9]])
z2=x2+y2
```

3. 파이토치 기초 문법

❖ 텐서의 형태 변환

- view함수() : 텐서의 행태(shape)를 변경

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

print(x.size()) # 결과: torch.size([3,2,2])
```

```
print(x.view(3,4))
print(x.view(3,1,4))
print(x.view(-1))
print(x.view(3, -1)
print(x.view(-1,1,4)
print(x.view(3,2,2,-1))
```

```
print(x.reshape(3,4))
print(x.reshape(3,1,4))
print(x.reshape(-1))
print(x.reshape(3, -1)
print(x.reshape(-1,1,4)
print(x.reshape(3,2,2,1))
```

3. 파이토치 기초 문법

❖ 텐서의 형태 변환

- Squeeze()함수 : 차원의 크기가 1인 차원 제거
- Unsqueeze()함수 : Squeeze() 반대 함수로 Unsqueeze()로 지정된 차원의 인덱스에 차원의 크기가 1인 차원 삽입

```
x = torch.FloatTensor([[[1, 2], [3, 4]]])
print(x.size())  #결과 : torch.Size([1, 2, 2])
```

```
print(x.squeeze())
print(x.squeeze().size())
print(x.squeeze(0).size())
print(x.squeeze(1).size())
```

```
x = torch.FloatTensor([[1, 2],[3, 4]])
print(x.size()) # 결과 : torch.Size([2, 2])
```

```
print(x.unsqueeze(2))
print(x.unsqueeze(-1))
print(x.reshape(2, 2, -1))
```

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- 인덱싱과 슬라이싱

```
x = torch.FloatTensor([[[1, 2],
                        [3, 4]],
                       [[5, 6],
                        [7, 8]],
                       [[9, 10],
                        [11, 12]]])
print(x.size())
# 결과 : torch.Size([3, 2, 2])
```

```
print(x[0])
print(x[0, :])
print(x[0, :, :])

print(x[-1])
print(x[-1, :])
print(x[-1, :, :])

print(x[:, 0, :])

print(x[1:3, :, :].size())
print(x[:, :1, :].size())
print(x[:, :-1, :].size())
```

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- Split 함수(), chunks() 함수

```
x = torch.FloatTensor(10, 4)

splits = x.split(4, dim=0)

for s in splits:
    print(s.size())
```

결과 :

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

```
x = torch.FloatTensor(8, 4)

chunks = x.chunk(3, dim=0)

for c in chunks:
    print(c.size())
```

결과 :

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

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- Index_Select 함수

```
x = torch.FloatTensor([[[1, 1],
                        [2, 2]],
                       [[3, 3],
                        [4, 4]],
                       [[5, 5],
                        [6, 6]]])
indice = torch.LongTensor([2, 1])
print(x.size())
```

결과: torch.Size([3, 2, 2])

```
y = x.index_select(dim=0, index=indice)

print(y)
print(y.size())
```

결과 :

```
tensor([[[5., 5.],
         [6., 6.]],

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

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- Concatenate 함수(cat())

```
x = torch.FloatTensor([[1, 2, 3],
                       [4, 5, 6],
                       [7, 8, 9]])
y = torch.FloatTensor([[10, 11, 12],
                       [13, 14, 15],
                       [16, 17, 18]])

print(x.size(), y.size())
```

결과 :

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

```
z = torch.cat([x, y], dim=0)
print(z)
print(z.size())
```

결과 :

```
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.],
        [ 7., 8., 9.],
        [10., 11., 12.],
        [13., 14., 15.],
        [16., 17., 18.]])
torch.Size([6, 3])
```

```
z = torch.cat([x, y], dim=-1)
print(z)
print(z.size())
```

결과 :

```
tensor([[ 1., 2., 3., 10., 11., 12.],
        [ 4., 5., 6., 13., 14., 15.],
        [ 7., 8., 9., 16., 17., 18.]])
torch.Size([3, 6])
```

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- stack()함수

```
x = torch.FloatTensor([[1, 2, 3],
                       [4, 5, 6],
                       [7, 8, 9]])
y = torch.FloatTensor([[10, 11, 12],
                       [13, 14, 15],
                       [16, 17, 18]])
```

```
z = torch.stack([x, y])
print(z)
print(z.size())
```

결과:

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

        [[10., 11., 12.],
         [13., 14., 15.],
         [16., 17., 18.]]])
torch.Size([2, 3, 3])
```

```
z = torch.stack([x, y], dim=-1)
print(z)
print(z.size())
```

결과 :

```
tensor([[[ 1., 10.],
         [ 2., 11.],
         [ 3., 12.]],

        [[ 4., 13.],
         [ 5., 14.],
         [ 6., 15.]],

        [[ 7., 16.],
         [ 8., 17.],
         [ 9., 18.]]])
torch.Size([3, 3, 2])
```

3. 파이토치 기초 문법

❖ 텐서 자르기 & 붙이기

- stack()함수

```
# z = torch.stack([x, y])
z = torch.cat([x.unsqueeze(0), y.unsqueeze(0)], dim=0)
print(z)
print(z.size())

결과 :
tensor([[[ 1., 2., 3.],
         [ 4., 5., 6.],
         [ 7., 8., 9.]],

        [[10., 11., 12.],
         [13., 14., 15.],
         [16., 17., 18.]]])
torch.Size([2, 3, 3])
```

```
result = []
for i in range(5):
    x = torch.FloatTensor(2, 2)
    result += [x]

result = torch.stack(result)
result.size()

결과 : torch.Size([5, 2, 2])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- Expand 함수: 차원의 크기가 1인 차원을 원하는 크기로 늘려 줌

```
x=torch.FloatTensor([[[1,2]],[[3,4]]])
print(x.size())

결과 : torch.Size([2, 1, 2])
```

```
y=x.expand(2, 3, 2)
print(y)

결과 :
tensor([[[1., 2.],
         [1., 2.],
         [1., 2.]],

        [[3., 4.],
         [3., 4.],
         [3., 4.]]])
```

```
y=torch.cat([x]*3, dim=1)
print(y)

결과 :
tensor([[[1., 2.],
         [1., 2.],
         [1., 2.]],

        [[3., 4.],
         [3., 4.],
         [3., 4.]]])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- Random Pernutation 함수
- 함수 명 : randperm(n) : 1~ n-1 범위의 숫자를 n개 랜덤 수열 생성

```
x=torch.randperm(10) #
print(x)
print(x.size())
```

결과 :

```
tensor([3, 7, 1, 6, 2, 0, 5, 8, 4, 9])
torch.Size([10])
```

```
x=torch.randperm(3**3).reshape(3,3,-1)
print(x)
print(x.size())
```

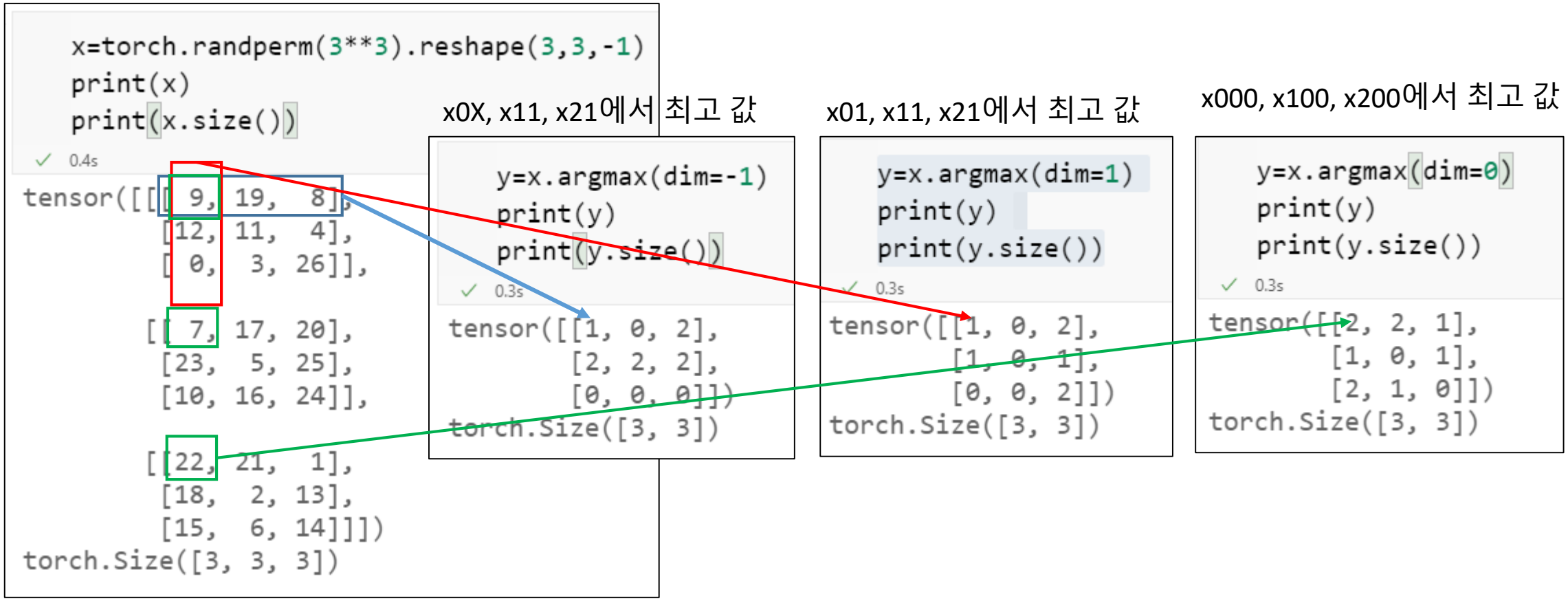
결과 :

```
tensor([[[ 1, 12, 18],
         [ 5, 10, 14],
         [16, 23,  3]],

        [[22, 19, 15],
         [ 7,  6,  9],
         [17,  2, 24]],

        [[ 4, 20, 21],
         [13, 25, 26],
         [ 8,  0, 11]]])
torch.Size([3, 3, 3])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- Argument Max 함수 : argmax(dim=n) (0:1차원, 1:2차원)



3. 파이토치 기초 문법

❖ 유용한 함수들

- Top-k 함수 : 가장 큰 k개 값과 index 반환

```
values,indices=torch.topk(x,k=1, dim=-1)
print(values)
print(values.size())
print(indices)
print(indices.size())
```

```
tensor([[[18],
         [14],
         [23]],

        [[22],
         [ 9],
         [24]],

        [[21],
         [26],
         [11]]])
torch.Size([3, 3, 1])
```

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

        [[0],
         [2],
         [2]],

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

```
values,indices=torch.topk(x,k=2, dim=-1)
print(values)
print(values.size())
print(indices)
print(indices.size())
```

```
tensor([[[18, 12],
         [14, 10],
         [23, 16]],

        [[22, 19],
         [ 9,  7],
         [24, 17]],

        [[21, 20],
         [26, 25],
         [11,  8]]])
torch.Size([3, 3, 2])
```

```
tensor([[[2, 1],
         [2, 1],
         [1, 0]],

        [[0, 1],
         [2, 0],
         [2, 0]],

        [[2, 1],
         [2, 1],
         [2, 0]]])
torch.Size([3, 3, 2])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- Top-k 함수 : 가장 큰 k개 값과 index 반환(sort 기능 활용)

```
_,indices=torch.topk(x, k=2, dim=-1)
print(indices.size())
```

```
# sort 기능
target_dim=-1
values, indices=torch.topk(x, k=x.size(target_dim), largest=True)
print(values)
```

결과

```
tensor([[[18, 12,  1],
         [14, 10,  5],
         [23, 16,  3]],

        [[22, 19, 15],
         [ 9,  7,  6],
         [24, 17,  2]],

        [[21, 20,  4],
         [26, 25, 13],
         [11,  8,  0]]])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- Masked Fill 함수 : 텐서 내에 원하는 부분 값만 변경

```
x=torch.FloatTensor([i for i in range(3**2)]).reshape(3, -1)
print(x)
print(x.shape)
```

```
tensor([[0., 1., 2.],
        [3., 4., 5.],
        [6., 7., 8.]])
torch.Size([3, 3])
```

```
mask=x>4
print(mask)
```

```
tensor([[False, False, False],
        [False, False,  True],
        [ True,  True,  True]])
```

```
y=x.masked_fill(mask, value=-1)
print(y)
```

```
tensor([[ 0.,  1.,  2.],
        [ 3.,  4., -1.],
        [-1., -1., -1.]])
```

3. 파이토치 기초 문법

❖ 유용한 함수들

- ones(), zeros(), ones_like(), zeros_like()

```
print(torch.ones(2,3))
print(torch.zeros(2,3))

x=torch.FloatTensor([[1,2,3],[4,5,6]])
print(x)
print(x.size())

print(torch.ones_like(x))
print(torch.zeros_like(x))
```