파이토치로 기초 문법

MLP

1.4 파이토치 기초

- 오픈소스 딥러닝 프레임워크
- 씨아노, 카페, 텐서플로와 달리 테이프기반 자동 미분 방식 구현
  - 계산 그래프를 동적으로 정의하고 실행 가능, 복잡한 모델을 쉽게 만들 수 있음
    - 정적 계산 그래프 : 계산 그래프를 정의하고 컴파일한 다음 실행

      제품 시스템과 모바일 환경에서 매우 효율적이지만 연구/개발에서는 번거로울 수 있음
    - 동적 계산 그래프 : 좀 더 유연한 명령형 스타일의 개발을 지원하며,

      실행할 때마다 모델을 컴파일하지 않아도 됨

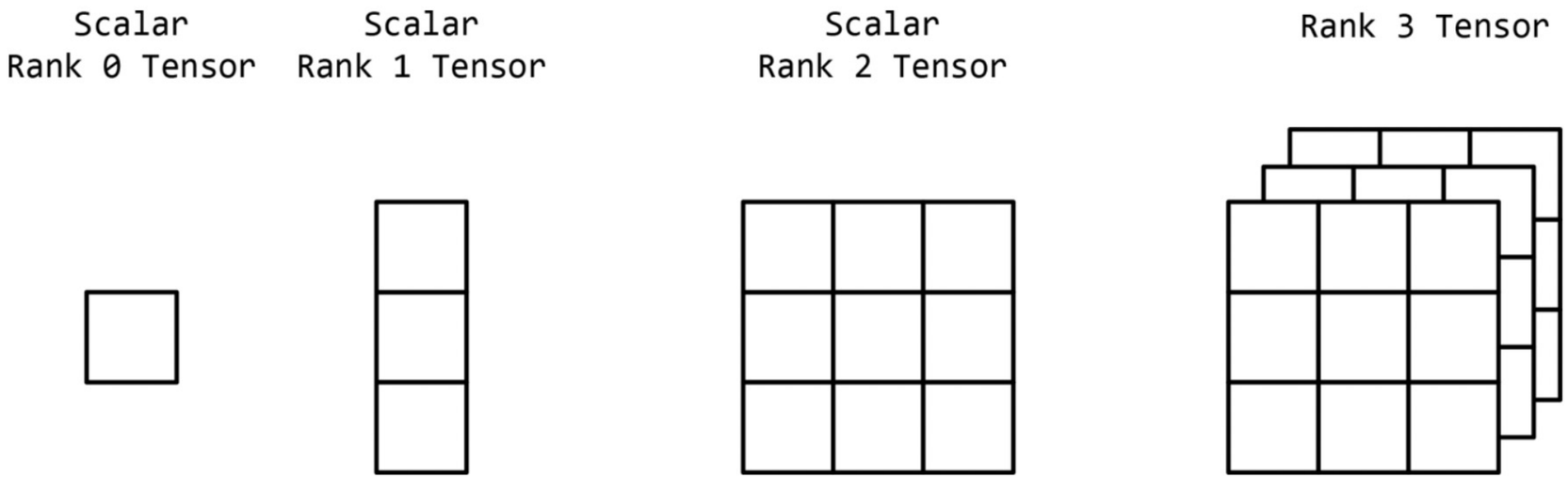
1.4 파이토치 기초

- 텐서

  - 다차원 데이터를 담은 수학 객체

  - 0차 텐서는 하나의 숫자 또는 스칼라, 1차 텐서는 숫자 배열 또는 벡터, 2차 텐서는 벡터의 배열 또는 행렬

  - n 차원 스칼라 배열로 일반화 가능



1.4 파이토치 기초

- 텐서 만들기

  - describe(x) : 텐서 타입, 차원, 값 같은 텐서의 속성을 출력하는 헬퍼 함수

```
def describe(x):
    print("타입: {}".format(x.type()))
    print("크기: {}".format(x.shape))
    print("값: \n{}".format(x))
```

  - 차원을 지정하여 랜덤하게 텐서 초기화

```
import torch
describe(torch.Tensor(2, 3))
```

```
executed in 7ms, finished 07:56:40 2023-02-01
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[2.9865e-34, 3.0875e-41, 2.9835e-34],
        [3.0875e-41, 1.1210e-43, 0.0000e+00]])
```

1.4 파이토치 기초

- 텐서 만들기

  - [0,1) 범위의 균등 분포 또는 표준 정규 분포에서 샘플링한 값으로 랜덤하게 초기화한 텐서 생성

```
1 import torch
2 describe(torch.rand(2, 3))  # 균등 분포
3 describe(torch.randn(2, 3)) #표준 정규 분포
```

executed in 8ms, finished 08:07:07 2023-02-01

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[0.7507, 0.1176, 0.7844],
        [0.1210, 0.6067, 0.1371]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[-0.3764, -0.2229,  1.9274],
        [-0.6607,  1.2755,  1.8517]])
```

1.4 파이토치 기초

- 텐서 만들기

  - fill()_메서드 사용해 특정값으로 채운 텐서 생성

    · 밑줄 문자(_)가 있는 파이토치 인-플레이스 메서드는 텐서값을 바꾸는 연산

    · 비슷하게 정규분포 normal_(), 균등 분포 uniform_()등의 인-플레이스 메서드가 있음

```
1 import torch
2 describe(torch.zeros(2, 3))
3 x = torch.ones(2, 3)
4 describe(x)
5 x.fill_(5)
6 describe(x)
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[0., 0., 0.],
        [0., 0., 0.]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[1., 1., 1.],
        [1., 1., 1.]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[5., 5., 5.],
        [5., 5., 5.]])
```

1.4 파이토치 기초

- 텐서 만들기

  - 파이썬 리스트를 사용해 텐서 생성

```
1 x = torch.Tensor([[1, 2, 3], [4, 5, 6]])
2 describe(x)
```

```
executed in 7ms, finished 08:22:03 2023-02-01

타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

  - 넘파이 배열을 사용해 텐서 생성

    · 텐서 타입이 기본 DoubleTensor로 생성됨

    → 넘파이 배열의 기본 데이터 타입이 float64이기 때문

```
1 import torch
2 import numpy as np
3 npy = np.random.rand(2, 3)
4 describe(torch.from_numpy(npy))
```

```
executed in 7ms, finished 08:28:11 2023-02-01

타입: torch.DoubleTensor
크기: torch.Size([2, 3])
값:
tensor([[0.4710, 0.3987, 0.4821],
        [0.9648, 0.4623, 0.4171]], dtype=torch.float64)
```

1.4 파이토치 기초

- 텐서 타입과 크기

  - torch.Tensor 생성자 사용시 기본 텐서 타입은 torch.FloatTensor

  - 텐서 타입은 초기화할 때 지정하거나 나중에 타입 캐스팅 메서드로 변경 가능

  - 초기화할 때 타입 지정 방법은 FloatTensor, LongTensor같은 특정 텐서 타입의 생성자 호출, torch.tensor() 메서드와 dtype 매개변수를 사용하는 방법이 있음

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

x = x.long()
describe(x)

x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]], dtype=torch.int64)
describe(x)

x = x.float()
describe(x)
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[1, 2, 3],
        [4, 5, 6]])
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[1, 2, 3],
        [4, 5, 6]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

1.4 파이토치 기초

- 텐서 타입과 크기

  - 텐서 객체의 shape 속성과 size() 메서드를 사용해 텐서의 차원 확인 가능

  - 파이토치 코드를 디버깅할 때는 텐서 크기를 조사하는 것이 중요

```
1 x.size()
```

```
executed in 5ms, finished 08:41:13 2023-02-01
```

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

1.4 파이토치 기초

- 텐서 연산

  - 일반적인 프로그래밍 언어처럼 +, -, *, /를 사용해 연산 수행 가능

  - 연산자 대신 .add() 같은 함수도 사용할 수 있음

```python
import torch
x = torch.randn(2, 3)
describe(x)

describe(torch.add(x, x))

describe(x + x)
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[ 0.5833,  0.2415,  0.3101],
        [ 0.6866, -0.8092,  0.2188]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[ 1.1666,  0.4831,  0.6202],
        [ 1.3731, -1.6184,  0.4376]])
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[ 1.1666,  0.4831,  0.6202],
        [ 1.3731, -1.6184,  0.4376]])
```

1.4 파이토치 기초

- 텐서 연산

  - 텐서의 특정 차원에 적용할 수 있는 연산

```python
import torch
x = torch.arange(6)
describe(x)

x = x.view(2, 3)
describe(x)

describe(torch.sum(x, dim=0))

describe(torch.sum(x, dim=1))

describe(torch.transpose(x, 0, 1))
```

```
타입: torch.LongTensor
크기: torch.Size([6])
값:
tensor([0, 1, 2, 3, 4, 5])
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[0, 1, 2],
        [3, 4, 5]])
타입: torch.LongTensor
크기: torch.Size([3])
값:
tensor([3, 5, 7])
타입: torch.LongTensor
크기: torch.Size([2])
값:
tensor([ 3, 12])
타입: torch.LongTensor
크기: torch.Size([3, 2])
값:
tensor([[0, 3],
        [1, 4],
        [2, 5]])
```

· arrange() : 0에서 시작해서 지정한 값 이전까지 1씩 증가하는 텐서 생성

· view() : 동일한 데이터를 공유하는 새로운 텐서 생성

1.4 파이토치 기초

- 인덱싱, 슬라이싱, 연결

  - 넘파이와 유사하게 인덱싱, 슬라이싱 가능

```python
import torch

x = torch.arange(6).view(2, 3)
describe(x)

describe(x[:1, :2])

describe(x[0, 1])
```

```
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[0, 1, 2],
        [3, 4, 5]])
타입: torch.LongTensor
크기: torch.Size([1, 2])
값:
tensor([[0, 1]])
타입: torch.LongTensor
크기: torch.Size([])
값:
1
```

1.4 파이토치 기초

- 인덱싱, 슬라이싱, 연결

  - 복잡한 인덱싱, 연속적이지 않은 텐서 인덱스 참조

    · index_select()가 반환하는 텐서는 dim에 지정한 차원의 크기가 index에 지정한 텐서의 길이와 같음

      나머지 차원은 원본 텐서와 크기가 같음

```
indices = torch.LongTensor([0, 2])
describe(torch.index_select(x, dim=1, index=indices))

indices = torch.LongTensor([0, 0])
describe(torch.index_select(x, dim=0, index=indices))

row_indices = torch.arange(2).long()
col_indices = torch.LongTensor([0, 1])
describe(x[row_indices, col_indices])
```

```
타입: torch.LongTensor
크기: torch.Size([2, 2])
값:
tensor([[0, 2],
        [3, 5]])
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[0, 1, 2],
        [0, 1, 2]])
타입: torch.LongTensor
크기: torch.Size([2])
값:
tensor([0, 4])
```

1.4 파이토치 기초

- 인덱싱, 슬라이싱, 연결

  - 내장 함수로 차원을 지정하여 텐서를 연결

```python
import torch

x = torch.arange(6).view(2,3)
describe(x)

describe(torch.cat([x, x], dim=0))

describe(torch.cat([x, x], dim=1))

describe(torch.stack([x, x]))
```

```
타입: torch.LongTensor
크기: torch.Size([2, 3])
값:
tensor([[0, 1, 2],
        [3, 4, 5]])
타입: torch.LongTensor
크기: torch.Size([4, 3])
값:
tensor([[0, 1, 2],
        [3, 4, 5],
        [0, 1, 2],
        [3, 4, 5]])
타입: torch.LongTensor
크기: torch.Size([2, 6])
값:
tensor([[0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5]])
타입: torch.LongTensor
크기: torch.Size([2, 2, 3])
값:
tensor([[[0, 1, 2],
         [3, 4, 5]],

        [[0, 1, 2],
         [3, 4, 5]]])
```

1.4 파이토치 기초

- 인덱싱, 슬라이싱, 연결

  - 파이토치는 텐서에서 매우 효율적인 선형 대수 연산 제공

    · 행렬 곱셈(mm), 역행렬(inverse, pinverse), 대각합(trace) 등

```python
import torch
x1 = torch.arange(6).view(2, 3).float()
describe(x1)

x2 = torch.ones(3, 2)
x2[:, 1] += 1
describe(x2)

describe(torch.mm(x1, x2))
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 3])
값:
tensor([[0., 1., 2.],
        [3., 4., 5.]])
타입: torch.FloatTensor
크기: torch.Size([3, 2])
값:
tensor([[1., 2.],
        [1., 2.],
        [1., 2.]])
타입: torch.FloatTensor
크기: torch.Size([2, 2])
값:
tensor([[ 3.,  6.],
        [12., 24.]])
```

1.4 파이토치 기초

- 텐서와 계산 그래프

  - 파이토치 tensor 클래스는 데이터와 대수연산, 인덱싱, 크기 변경 등 다양한 연산 캡슐화

    · 텐서의 requires_grad 매개변수를 True로 지정하면 그레이디언트 연산을 할 수 있는 텐서 생성

```python
import torch
x = torch.ones(2, 2, requires_grad=True)
describe(x)
print(x.grad is None)

y = (x + 2) * (x + 5) + 3
describe(y)
print(x.grad is None)

z = y.mean()
describe(z)
z.backward()
print(x.grad is None)
```

```
타입: torch.FloatTensor
크기: torch.Size([2, 2])
값:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
True
타입: torch.FloatTensor
크기: torch.Size([2, 2])
값:
tensor([[21., 21.],
        [21., 21.]], grad_fn=<AddBackward0>)
True
타입: torch.FloatTensor
크기: torch.Size([])
값:
21.0
False
```

1.4 파이토치 기초

- CUDA 텐서

  - GPU를 사용하려면 텐서를 GPU 메모리에 할당해야 함

  - CUDA API

    · NVIDIA 에서 만들었고, NVIDIA GPU에서만 사용할 수 있음

  - 내부적인 할당 방식만 다르고 CPU 텐서와 사용법이 같음

1.4 파이토치 기초

- CUDA 텐서

  - torch.cuda.is_available()로 GPU를 사용할 수 있는지 확인

  - torch.device()로 장치 이름을 가져온뒤 .to(device)로 텐서를 타깃 장치로 이동

```python
import torch
print(torch.cuda.is_available())

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

x = torch.rand(3, 3).to(device)
describe(x)
```

```
True
cuda
타입: torch.cuda.FloatTensor
크기: torch.Size([3, 3])
값:
tensor([[0.3753, 0.4631, 0.4456],
        [0.8755, 0.0060, 0.5917],
        [0.3439, 0.7928, 0.8748]], device='cuda:0')
```

1.4 파이토치 기초

- CUDA 텐서

  - CUDA 객체와 CUDA가 아닌 객체를 다루려면 두 객체가 같은 장치에 있는지 확인해야 함

```
y = torch.rand(3, 3)
x + y
```

executed in 30ms, finished 09:52:04 2023-02-01

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [41], in <cell line: 2>()
      1 y = torch.rand(3, 3)
----> 2 x + y

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and
cpu!
```

```
cpu_device = torch.device("cpu")
y = y.to(cpu_device)
x = x.to(cpu_device)
x + y
```

executed in 16ms, finished 09:52:42 2023-02-01

```
tensor([[1.3692, 1.2164, 1.1527],
        [1.3015, 0.9541, 1.3979],
        [0.8240, 1.4793, 1.0815]])
```

  - GPU로 데이터를 넣고 꺼내는 작업은 비용이 많이 듦

    → 병렬 계산은 일반적으로 GPU에서 수행하고 최종 결과만 CPU로 정송하는 방식

  - CUDA 장치가 여럿이라면 CUDA_VISIBLE_DEVICES 환경변수를 사용

2 파이토치 튜토리얼

- 아래 튜토리얼 정도는 이해해야 수업 진행이 가능합니다!
- https://pytorch.org/tutorials/beginner/pytorch_with_examples.html
- https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

감사합니다.