13장 파이토치 구조 자세히 알아보기

~13.5

1. 파이토치의 주요 특징

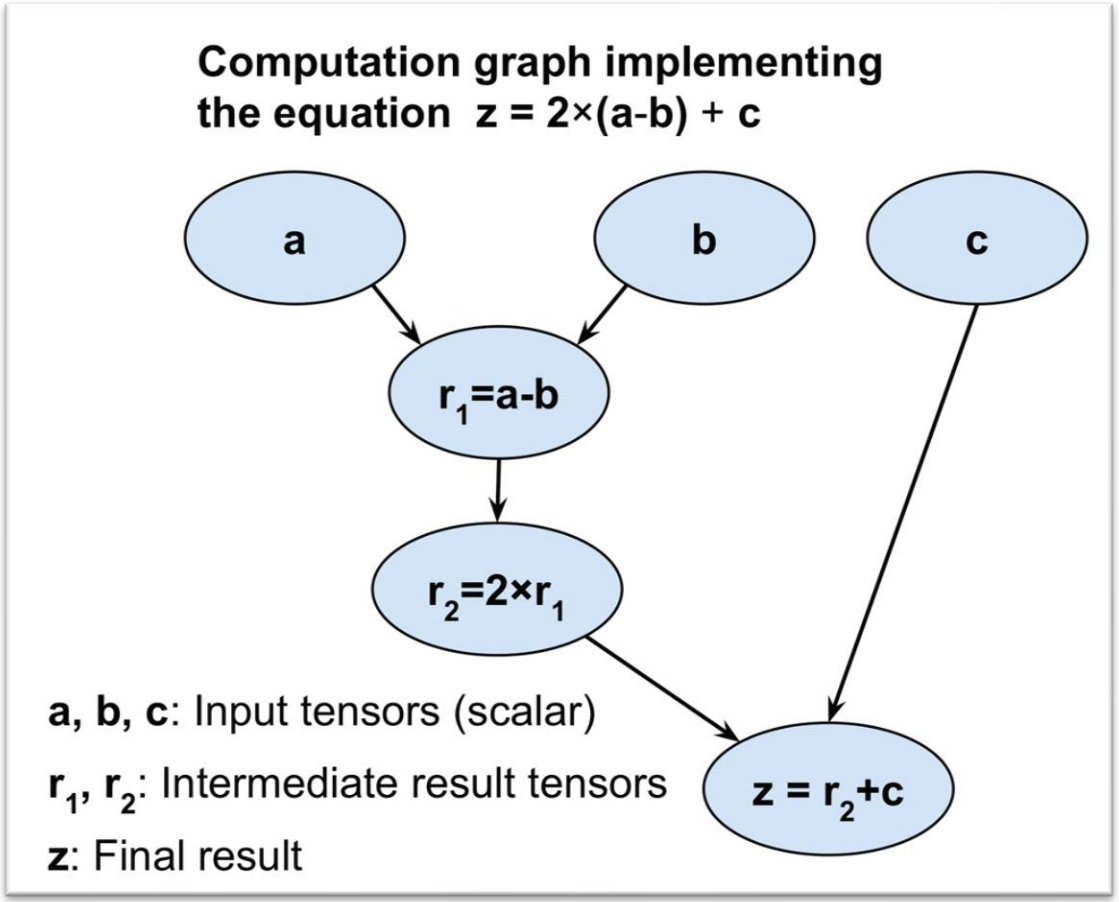
- PyTorch는 딥러닝을 위한 가장 인기 있는 프레임워크 중 하나.
- 2016년에 처음 출시, 오픈 소스 라이브러리로 페이스북에서 자금과 지원을 받다가 현재는 리눅스 재단으로 이관.
- 정적 그래프에 비해 유연성이 뛰어난 동적 그래프 사용.
- 동적 그래프는 코드를 실행하는 순간
  계산 그래프가 자동으로 만들어짐

2. 파이토치의 계산 그래프

- 파이토치는 DAG(Directed Acyclic Graph, 유향비순환 그래프)를 기반으로 계산을 수행하고, 이를 사용하여 그레이디언트를 계산함

- 데이터는 한 방향으로 흐르고(유향,방향이 있는),
  뒤 연산이 앞 연산의 결과에 의존하지만,
  다시 앞 연산에 영향을 주는 루프는 없다.

- 계산 그래프는 유향이고, 순환이 없어야 한다. 그래야 미분을 역방향으로 계산(역전파) 할 때 문제가 없음.



계산 그래프의 작동 방식

3. 모델 파라미터를 저장하고 업데이트 하기 위한 파이토치 텐서 객체

- 파이토치에서 텐서를 생성할때 requires_grad를 True로 지정하면 그레이디언트를 계산하고 모델의 파라미터를 저장하고 업데이트 할 수 있다.

- requires_grad_() 메서드를 호출하여 이 값을 True로 설정할 수도 있다.

Note : 밑줄(_)이 붙은 연산자들은 전부 in-place 연산자임.

```
a = torch.tensor(3.14, requires_grad=True)
b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
print(a)
print(b)
```

```
tensor(3.1400, requires_grad=True)
tensor([1., 2., 3.], requires_grad=True)
```

```
w = torch.tensor([1.0, 2.0, 3.0])

print(w.requires_grad)
```

```
False
```

```
[5] w.requires_grad_()

    print(w.requires_grad)
```

```
True
```

3. 모델 파라미터를 저장하고 업데이트 하기 위한 파이토치 텐서 객체

- 신경망 모델의 초기 가중치가 모두 동일하다면

- 각 뉴런의 출력은 전부 같아지고,
  그 결과, 오차(error)도 같고,
  역전파로 전달되는 gradient(기울기)도 전부 똑같다.
  결국 업데이트되는 방향도 전부 같아짐.

⇒ 이 경우를 대칭적(symmetrical) 이라 함.

- 대칭성을 깨기 위해 파라미터를 랜덤한 가중치로 초기화 해야함.
- 딥러닝 개발 초창기에 무작위한 균등 분포나 정규 분포를 사용한 가중치 초기화는 나쁜 성능을 만든다고 관측됨
- 세이비어와 요슈아는 이 문제에 안정적인 초기화 방법을 제안함.

◦ Xavier Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

(n_{in} : 이전 layer(input)의 노드 수, n_{out} : 다음 layer의 노드 수)

◦ Xavier Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \quad +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

(n_{in} : 이전 layer(input)의 노드 수, n_{out} : 다음 layer의 노드 수)

세이비어와 요슈아가 제안한 가중치 분포

3. 모델 파라미터를 저장하고 업데이트 하기 위한 파이토치 텐서 객체

```
import torch.nn as nn


torch.manual_seed(1)
w = torch.empty(2, 3)
nn.init.xavier_normal_(w)
print(w)
```

```
tensor([[ 0.4183,  0.1688,  0.0390],
        [ 0.3930, -0.2858, -0.1051]])
```

세이비어와 요슈아가 제안한 가중치 분포를 실현하는 nn.init 모듈의 xavier_normal_() 함수

Note : 밑줄(_)이 붙은 연산자들은 전부 in-place 연산자임.

```
class MyModule(nn.Module):
      def __init__(self):
          super().__init__()
          self.w1 = torch.empty(2, 3, requires_grad=True)
          nn.init.xavier_normal_(self.w1)
          self.w2 = torch.empty(1, 2, requires_grad=True)
          nn.init.xavier_normal_(self.w2)
```

nn.Module을 상속한 클래스 안에서 Tesnsor 객체의 가중치 설정하기

4. 자동 미분으로 그레이디언트 계산

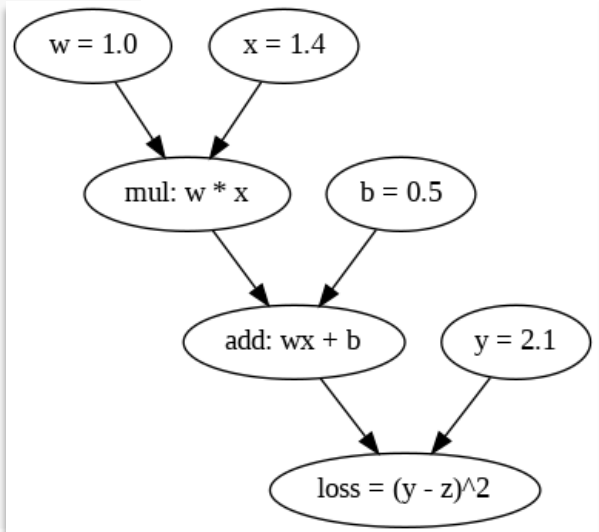
- 파이토치는 자동미분을 지원.
- 이는 중첩된 함수(합성함수)의 그레이디언트를 계산하기 위해 연쇄법칙을 구현한 것으로 생각할 수 있음.

- PyTorch는 자동으로 계산 그래프를 구성하고, 이를 통해 의존성이 있는 연산에 대해 기울기를 계산하는 기능을 제공하며,
- Tensor 객체에 정의된 backward 메서드를 호출하여 의존성이 있는 텐서에 대한 그레이디언트 계산가능.

```
w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.5, requires_grad=True)

x = torch.tensor([1.4])
y = torch.tensor([2.1])


z = torch.add(torch.mul(w, x), b)

loss = (y-z).pow(2).sum()
loss.backward()

print('dL/dw : ', w.grad)
print('dL/db : ', b.grad)
```

```
dL/dw :  tensor(-0.5600)
dL/db :  tensor(-0.4000)
```

loss 의 그레디언트 계산



loss 의 계산그래프

5. torch.nn 모듈을 사용하여 일반적인 아키텍처 구현하기

- nn.Module 말고도 nn.Sequential 클래스를 사용하여 비슷한 구조의 모델을 간단히 만들 수 있다.
- 직관적으로 층과 활성화 함수를 순서대로 (sequential하게) 쌓아 구성할 수 있다.
- Linear 는 완전연결층(한 층의 모든 뉴런이 그 다음 층의 모든 뉴런과 연결된 상태)을 의미

```
model = nn.Sequential(
      nn.Linear(4, 16),
      nn.ReLU(),
      nn.Linear(16, 1),
      nn.Sigmoid()
                    )
model #nn.Sequential 클래스
```

```
Sequential(
  (0): Linear(in_features=4, out_features=16, bias=True)
  (1): ReLU()
  (2): Linear(in_features=16, out_features=1, bias=True)
  (3): Sigmoid()
)
```

5. torch.nn 모듈을 사용하여 일반적인 아키텍처 구현하기

- nn.Sequential 클래스로 여러 개의 층을 가진 완전 연결 신경망을 만들 수 있다.

- 하지만 여러 개의 입력이나 출력을 가지거나 중간 가지(branch)가 있는 복잡한 모델을 만들 수 없다. 그래서 nn.Module이 필요하다.

- 사용자 정의 layer도 nn.Module을 상속받아 정의할 수 있다.