자동미분과 PyTorch Autograd

- 2023.11.21 작성

1. 목차

2. 개요

2.1. 왜 Autograd를 주제로 정했나요?

- 지난주 PyTorch 강의 들으면서, PyTorch의 핵심 모듈중 하나로 **Autograd**가 나왔어요.
- 근데 강의에선, **tensor.backward()** 하나로 설명이 끝나더라구요…?? [1]
- 굉장히 기대하면서 보다가 맥이 빠졌어요.
- 사용자의 **어떤 불편함**을 **어떤 설계적인 내용으로 해결**을 했는지를 기대하고 있었거든요.
- 그래서 직접 한 번 찾아봤어요!
- 내용은 다음과 같은 흐름으로 진행돼요.
- 먼저, 프레임워크에 대해 알아봤어요.
  - PyTorch는 프레임워크라고 하던데, **프레임워크가 정확히 무엇인지** 알아야 제공하는 기능에 대해서도 잘 이해할 수 있을 것 같았거든요.
  - 찾아보니, **라이브러리**라는 친구가 항상 짝지어 다니더라구요.
  - **PyTorch**가 왜 라이브러리가 아니라 **프레임워크**인지 찾아봤어요.
- 다음으로, **자동미분**에 대해 알아봤어요.
  - 딥러닝 모델 학습의 과정을 분리해 생각해보면서, 자동미분이 정확히 **어떤 부분**을 처리하는 기능인지 찾아봤어요.
  - 그리고 **예제 코드**를 보면서, **자동미분**이 정확히 **무엇**인지 알아봤어요.
- 마지막으로, PyTorch의 Autograd 에 대해 알아봤어요.
  - 강의에서 하나로 퉁쳤던 **backward()가 어떻게 동작**하는지 찾아봤어요.

3. 프레임워크와 라이브러리

3.1. 프레임워크가 무엇인가요?

> [2] **프로그래밍과 엔지니어링 프레임 워크란 무엇인가요?**
>
> 소프트웨어 엔지니어링 및 프로그래밍에서 프레임워크는 새로운 애플리케이션을 보다 효율적으로 개발할 수 있도록 하는 **재사용 가능한 소프트웨어 구성 요소의 모음**입니다. 모든 엔지니어링

> 분야에서 기존 개발 및 연구 결과를 재사용하는 것은 기본 원칙입니다. 예를 들어, 전기 엔지니어는 기존 전자 부품을 사용하여 새로운 장치를 만듭니다. 부품 제조업체는 부품의 사용성을 보장하기 위해 **사전에 정해진 표준과 규격**을 준수합니다. 마찬가지로 소프트웨어 프레임워크에는 특정 소프트웨어 **표준 및 프로토콜을 기반으로 재사용 가능한 코드 모듈**이 포함되어 있습니다. 또한 프레임워크는 특정 소프트웨어 아키텍처 규칙 또는 비즈니스 프로세스를 정의하고 적용할 수 있으므로 새로운 애플리케이션을 표준화된 방식으로 개발할 수 있습니다.
>
> https://aws.amazon.com/ko/what-is/framework/

- AWS가 알려주길, 프레임워크는 **사전에 정해진 규칙에 따라 개발한 재사용 가능한 소프트웨어 코드 모듈의 모음** 이래요.

> **소프트웨어 프레임워크 사용의 이점은 무엇인가요?**
>
> 소프트웨어 프레임워크는 개발자가 소프트웨어를 구축할 때 기존에 작업하는 방식을 바꿉니다. 소프트웨어 팀과 조직이 프로그래밍 프레임워크를 사용하면 여러 면에서 이점을 얻을 수 있습니다.
>
> **코드 품질 개선**
>
> 소프트웨어 프레임워크에는 높은 프로그래밍 표준에 맞게 설계된 소프트웨어 구성 요소가 포함되어 있습니다. 개발자는 기본 코드에 영향을 미치는 버그가 줄어들 것이라는 확신을 가지고 소프트웨어 프레임워크를 사용할 수 있습니다. 또한 소프트웨어 프레임워크는 코드 가독성을 향상시키는 방식으로 구조화되어 있습니다. 프레임워크가 추상화하는 소프트웨어 워크플로에 대한 공통된 이해를 공유하면 소프트웨어 팀이 더 쉽게 협업할 수 있습니다.
>
> **개발 시간 단축**
>
> 소프트웨어 프레임워크는 프로그래밍 효율성을 향상시키고 조직은 이를 사용하여 제대로 작동하는 애플리케이션을 더 빠르게 릴리스할 수 있습니다. 개발자는 좋은 소프트웨어 프레임워크를 사용하여 기본 코딩 모듈 대신 비즈니스 로직을 처리하는 고급 코드를 작성하는 데 집중할 수 있습니다. 예를 들어 개발자는 오픈 소스 프레임워크를 사용하여 데이터베이스 액세스를 제공하고 이를 기반으로 전자 상거래 소프트웨어를 개발할 수 있습니다.
>
> 또한 프레임워크를 사용하여 개발자는 애플리케이션을 느리게 하거나 부풀릴 수 있는 중복된 코드를 작성하지 않아도 됩니다.
>
> **더 개선된 소프트웨어 보안**
>
> 코드베이스가 광범위하면 개발자가 코드 보안 문제를 감지하고 이에 대응하기가 어렵습니다. 반면, 좋은 소프트웨어 프레임워크는 개발자가 코드 및 데이터 보안을 더 쉽게 강화할 수 있도록 준비된 보안 체크포인트로 구성됩니다.
>
> **효율적인 코드 검토**
>
> 개발자는 애플리케이션을 릴리스하기 전에 여러 개발 단계에서 코드를 테스트합니다. 모든 소프트웨어 함수, API, 데이터 구조 및 모듈은 특정 코드 검토 요구 사항을 통과해야 합니다. 소프트웨어 팀은 프레임워크를 사용하여 포괄적인 테스트 사례 및 코드 커버리지로 애플리케이션을 검증할 수 있습니다. 또한 개발자는 잘 구조화된 프레임워크에서 코드 문제를 더 쉽게 디버깅하고 수정할 수 있습니다.

> **개발 유연성**
>
> 개발자는 소프트웨어 프레임워크를 사용하여 주요 소프트웨어 변경 사항을 구현할 때 더 민첩하게 대응할 수 있습니다. 목표에 맞는 다양한 프레임워크를 교체하면서 프로젝트별 코드를 유지할 수 있습니다. 이렇게 하면 개발자가 수행해야 하는 코드 재작성 작업이 줄어듭니다. 예를 들어 기존 기계 학습(ML) 프레임워크를 보다 강력한 프레임워크로 대체하여 이미지 인식 애플리케이션을 업그레이드할 수 있습니다.

- 프레임워크를 사용하면 **짧은 시간**에, **높은 품질의 코드**를 만들면서 **보안**과 **유연성**까지 챙길 수 있대요.
- 굉장히 좋은 친구인걸요?
- 근데, **어떻게** 이게 가능하죠?

> **프레임워크는 어떻게 작동하나요?**
>
> 프레임워크는 개발자가 소프트웨어 개발을 가속하여 프로덕션 배포에 이르는 데 도움이 되는 유연한 범위의 소프트웨어 구성 요소를 제공합니다. API, 코드 라이브러리, 디버거 및 컴파일러와 같은 프로그래밍 언어용으로 빌드된 리소스로 구성됩니다. 예를 들어 Ruby on Rails는 Ruby 언어로 개발된 웹 애플리케이션 프레임워크입니다.
>
> 다음은 일반적인 프레임워크 구성 요소에 대해 설명합니다.
>
> - API는 서로 다른 소프트웨어가 상호 이해할 수 있는 형식으로 통신할 수 있도록 하는 프로토콜입니다.
> - 코드 라이브러리는 개발자가 코드에 연결할 수 있는 재사용 가능한 소프트웨어 함수 모음입니다.
> - 컴파일러는 개발자가 소스 코드를 배포 가능한 애플리케이션 파일로 변환하는 데 사용하는 소프트웨어 도구입니다.
> - 디버거는 프로그래머가 코드에서 실수를 찾아 수정하는 데 도움이 되는 도구입니다.
>
> **제어 역전**
>
> 개발자는 소프트웨어 프레임워크의 일부를 애플리케이션의 구성 요소로 사용합니다. 프레임워크는 개발 속도를 높이는 데 필요한 리소스를 제공하지만 **애플리케이션의 절차적 흐름도 변경**합니다.
>
> 제어 역전(IoC)은 기존 제어 흐름과 비교하여 제어 흐름을 역전시키는 설계 원칙입니다. 흐름을 제어하고 재사용 가능한 라이브러리를 호출하는 애플리케이션 코드 대신, 기본 **애플리케이션은 프레임워크에 제어권을 넘겨줍니다**. 그런 다음 프레임워크는 다양한 소프트웨어 메커니즘을 통해 **애플리케이션 코드에 대한 추가 지원과 지침을 제공**합니다. 그 결과 소프트웨어 함수와 클래스가 느슨하게 결합되어 소프트웨어 유지 관리 용이성, 유연성 및 확장성이 향상됩니다.

- **제어 역전(Inversion Of Control)** 을 통해 애플리케이션의 **전체 흐름을 프레임워크가 컨트롤**하면서 정확하게 동작하는 **코드 작성의 방향성**도 잡아주는 거군요.
- 그러면 목적지를 입력하면 알아서 길을 찾아주는 🧭**네비게이션** 같은 친구네요!
  - 네비게이션이 없던 시절엔 🗺️지도를 보면서 운전했대요 [3]

- 근데 프레임워크 구성 요소에 **라이브러리** 라는 친구가 나오는데, 이 친구도 **재사용 가능한 소프트웨어 함수의 모음**이래요.
- 그러면 라이브러리도 프레임워크랑 같은거 아닌가요?

3.2. 프레임워크와 라이브러리는 어떻게 다른가요?

> **프레임워크와 라이브러리 비교**
>
> 프레임워크와 라이브러리는 모두 개발자가 애플리케이션을 더 효율적으로 빌드할 수 있도록 다른 사람이 작성한 재사용 가능한 코드입니다. 그러나 **라이브러리**는 **필요할 때 애플리케이션 코드가 호출하는** 유틸리티 또는 함수의 모음입니다. 라이브러리는 작업별로 다릅니다. 예를 들어 이미 작성된 코드로 ML을 구현할 수 있습니다. 라이브러리는 애플리케이션의 도우미이자 도구 역할을 합니다.
>
> 반대로 **프레임워크**는 애플리케이션 개발을 지시하는 구조적 청사진입니다. 개발자가 세부 사항을 입력할 수 있는 골격을 제공합니다. 즉, 아키텍처를 준수하여 **동작의 일부를 사용자 지정**할 수 있습니다. 개발자는 프레임워크의 규칙과 구조를 중심으로 애플리케이션과 아키텍처를 구성해야 합니다. 제어 흐름은 프레임워크에도 전달되며, 프레임워크는 필요한 경우 내부적으로 라이브러리를 호출할 수 있습니다.

- 라이브러리는 애플리케이션 코드(=개발자)가 호출하는 **수동적인 친구**이고, 프레임워크는 애플리케이션 코드(=개발자)를 호출하는 **능동적인 친구**인거군요!

3.3. PyTorch는 왜 프레임워크인가요?

- 프레임워크가 능동적인 친구를 뜻하는 건 알았어요.

- 그럼 PyTorch 는 우리한테 어떤걸 해주는 친구인거예요?

> [4] PyTorch는 딥러닝 모델을 만들고 학습시키기 위한 고수준의 추상화와 제어의 흐름을 가지고 있습니다. **모델의 구조, 학습 알고리즘, 데이터 로딩** 등의 부분에서 프레임워크가 개발자에게 제어의 주도권을 주고 있습니다. 따라서 PyTorch는 프레임워크로 분류됩니다.

- **PyTorch**는 **모델의 구조 생성, 학습 알고리즘 실행, 데이터 로딩 등 요청한 부탁을 대신해주는 심부름꾼 친구**군요!
- 그럼 **자동미분**도 PyTorch한테 요청하면 대신해주는걸 **기대할 수 있는 부탁**인가봐요!
- 이제 자동미분을 부탁해봐야겠어요!!!
- 근데….. 자동미분이… 뭐였죠….?? [5]

4. 자동미분

- PyTorch가 부탁하면 들어주는 착한 친구라는 건 알았어요.
- 자동미분이란걸 해야해서 부탁하고 싶은데.. **자동미분**이 뭔지 몰라서 **언제 부탁해야 할지**를 모르겠어요..;;
- 자동미분이 뭐예요..??

4.1. 자동미분이… 뭐죠?

- 미분은 뭔지 알아요! 고등학교때 배웠어요!!
  - 음, 오래돼서… 정확히 기억은 안나지만… 아래 같은 공식은 기억나요!

$$f(x) = ax \ : \ f'(x) = a$$
$$f(x) = x^n \ : \ f'(x) = nx^{n-1}$$
$$f(x) = log(x) \ : \ f'(x) = \frac{1}{x}$$
$$f(x) = e^x \ : \ f'(x) = e^x$$

- 미분을 떠올리고보니 자동미분은 **미분을 자동으로 해주는**건가봐요!
- 근데 이걸 **PyTorch에 언제 요청**해야 하는 거예요??

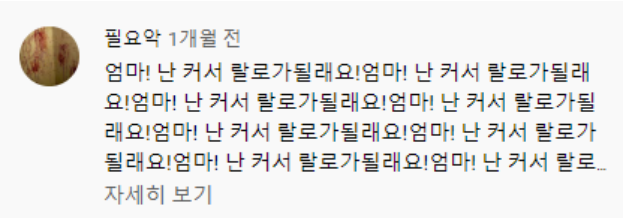
4.2. 자동미분을 언제 요청해야 하나요?

- PyTorch는 **딥러닝 모델의 학습을 대신 처리해주는 친구**예요.

- 그러고보니 **딥러닝 모델 학습**할 때 **미분**을 했었던것 같아요!
- 언제였지.... forward로 loss 를 구해주고.. 각 **노드별로 loss에 대한 영향도**를 구할 때 였었던 것 같아요!
- 딥러닝 모델 학습 과정을 한 번 단계별로 나열해 봐야겠어요!
    - 딥러닝 모델을 생성하고
    - 파라미터들을 초기화해준 뒤
    - 학습 데이터 값을 모델에 입력해서
    - 예측값을 계산한 다음
    - 예측값과 답을 비교해서 오차를 계산해요.
    - **그런 다음 해당 오차를 각 노드의 함수로 미분하면서**
    - **해당 함수에 입력됐던 값들로 미분값을 구해줘요.**
    - **계산한 미분값을 예측값 계산시 값을 전달해준 노드들로 반대로 넘겨주면서**
    - **최종 입력값의 미분값까지 계산해요.**
    - Optimizer 로 Parameter 값을 수정해요.
- 근데 나도 미분할 줄 아는데.. 직접하면 안되는거예요???

4.3. 직접하면 안돼요?

- 나는 **배려심**이 넘치는 친구로서, 이미 많은 일을 하는 **PyTorch를 도와주고 싶어요!**
- 난 내가 **직접 미분**을 할래요!
    - 엄마! 난 커서 미분을 할래요! [6]



- 엄마한테 얘기했더니 **컴퓨터로 미분하는 법**을 알려주셨어요!

4.4. 엄마가 알려준 컴퓨터를 활용한 미분 [7]

4.4.1. 변수와 함수

- 코드는 **재사용**이 가능하도록 **OOP(Object Oriented Programming)** 적으로 작성하는게 좋다고 우리 엄마가 그랬어요!
    - OOP적으로 코드를 작성하면, 코드간에 **결합도가 낮아**지고 **응집도는 높아**져서 수정해야 하는 부분만 집중할 수 있대요!
        - 엄마가 아끼시는 진주목걸이📿랑 친구가 준 구슬목걸이랑 섞어서 엮어놨다가 엄마한테 혼났어요... 여러분도 조심하세요...

```
class Variable:
    def __init__(self, data):
        self.data = data
```

- 먼저 **변수**를 의미하는 **Variable 클래스**를 만들었어요!

- 변수는 **값을 담을 수 있는 상자** 예요.
- data 는 상자가 담고 있는 **값**을 의미해요.

```
class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(y)
        return output

    def forward(self, x):
        raise NotImplementedError()
```

- 다음은 **함수**를 나타내는 **Function 클래스**예요!
  - Variable 형태의 input 을 받아서 정의된 연산 수행 결과를 Variable 형태로 반환해줘요.
  - 이때 다양한 **연산 형태를 유연하게 정의**하기 위해 forward메소드는 **추상화**한 상태로, **상속받는 클래스에서 재정의** 하도록 했어요!

```
class Square(Function):
    def forward(self, x):
         return x ** 2 # x 는 Variable에 담긴 값을 의미

class Exp(Function):
    def forward(self, x):
         return np.exp(x)
```

- 위에서 정의한 Function클래스를 상속한 Square 클래스와 Exp 클래스를 생성했어요!
  - 각각 아래 함수 형태를 나타내요!

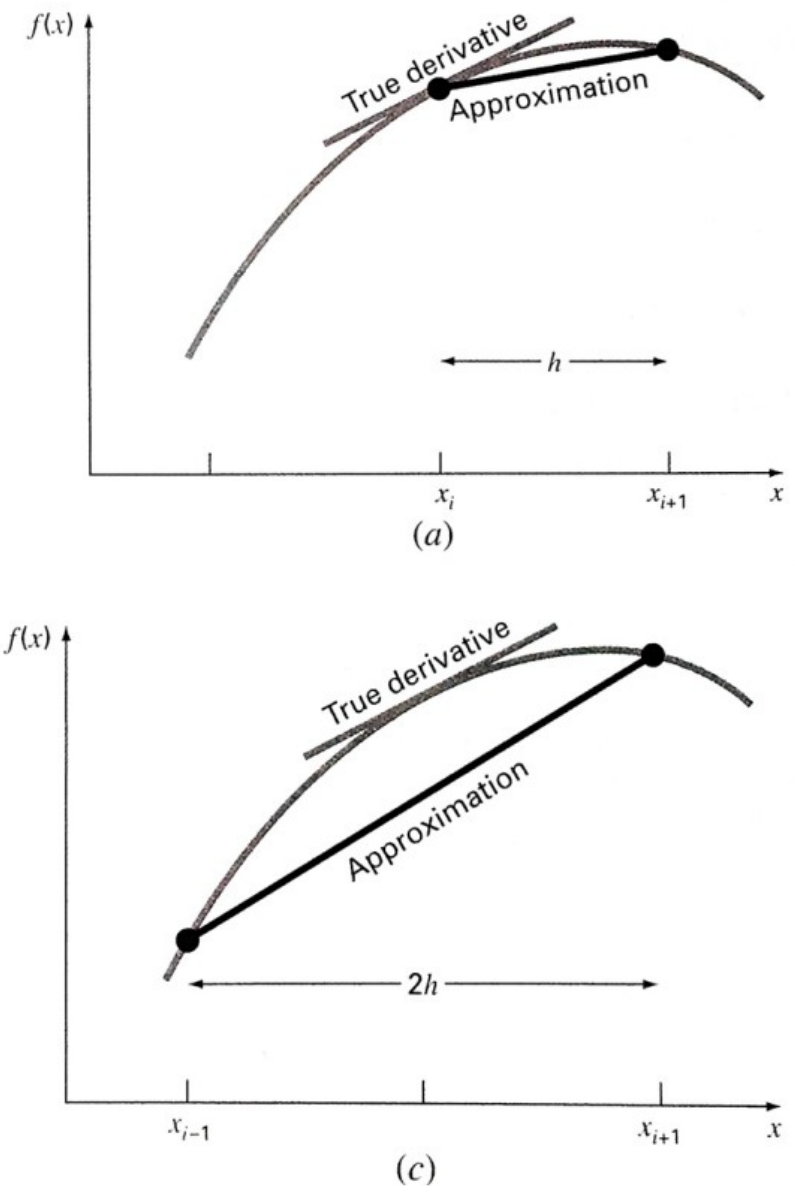
$$Square : f(x) = x^2$$
$$Exp : f(x) = e^x$$

4.4.2. 수치 미분

- 이제 함수와 변수를 입력하면, 해당 변수 값에서의 **미분값을 계산하는 메소드**를 만들어요!

```
def numerical_diff(f: Function, x: Variable, eps=1e-4):
    x0 = Variable(x.data - eps)
    x1 = Variable(x.data + eps)
    y0 = f(x0)
    y1 = f(x1)
    return (y1.data - y0.data) / (2 * eps)
```

- 이러한 방식을 **수치 미분**이라고 한대요!
- 위 메소드는 **중앙 차분** 이라는 방식으로 구현한 코드예요.
  - 일반적인 **전진 차분** 방식에 비해 더 정확하게 근사가 가능하대요! [8]



자료출처 : Numerical Methods for Engineers 6th edition /Chapra, Canale / McGRAW-HILL

- 하지만 위 방식은 모든 (함수, 입력값) 조합에 대해 각각 계산을 해줘야해서, 딥러닝 모델과 같이 굉장히 많은 (함수, 입력값) 조합이 존재하는 모델에선 **연산량이 너무 많기 때문에** 사용하기가 어려워요!
- 다른 방식으론 **기호 미분**이 있대요!
  - 이 방식은 미분 공식을 이용해, 각 Function 클래스의 **도함수를 나타내는 클래스**를 계산해서 변수값에 해당하는 함수값을 구하는 식으로 작동한대요.
    - 입력 함수식을 **트리 구조**로 관리한다고 하네요.
  - 이 방식은 수식이 복잡해지면 트리가 굉장히 커져요!
  - 그리고 우리는 도함수가 필요한게 아니라 미분값이 필요하니까 배보다 배꼽이 더 커지는 방식이라고 엄마가 그랬어요!
  - 엄마도 까먹어서 코드는 없어요ㅎ

4.4.3. 자동 미분 with 수동 역전파

- 마지막 방식이 진정한 **자동 미분** 이래요!
- 미분의 **연쇄 법칙을 사용**하는 방식이예요!

```
class Variable:
    def __init__(self, data):
        self.data = data
        self.grad = None # 변수에 미분결과 저장

class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(y)
        self.input = input # 미분값 계산시 사용할 값 저장
        return output

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, x): # 미분값 계산하는 추상화 메소드
        raise NotImplementedError()

class Square(Function):
    def forward(self, x):
        return x ** 2

    def backward(self, gy): # 미분값 계산 메소드; gy는 이전 단계에서 넘어온 미분값
        x = self.input.data
        gx = 2 * x * gy
        return gx

class Exp(Function):
    def forward(self, x):
        return np.exp(x)

    def backward(self, gy): # 미분값 계산 메소드
        x = self.input.data
        gx = np.exp(x) * gy
        return gx
```

- 강조 표시한 부분이 새로 추가된 부분이예요!
  - 각 변수에 미분값 결과를 저장할 수 있도록 Variable 클래스에 **grad** 속성을 추가했어요!
  - Function 클래스에는 미분값을 계산하는 **backward()를 추상화**해두고, **Square와 Exp에서 실제 구현한 메소드를 추가**했어요.

```
A = Square()
B = Exp()
C = Square()

# 순전파 과정
x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b) # y = C(B(A(x)))

# 역전파 과정
y.grad = np.array(1.0)
b.grad = C.backward(y.grad)
a.grad = B.backward(b.grad)
x.grad = A.backward(a.grad)
print(x.grad)

3.297442541400256
```

- 위 코드는 🌟**역전파 순서에 맞춰 직접 호출하는 코드를 작성**해줘야 하네요..

    - 이러면 🌟**순서를 내가 기억해야 하잖아요**??? 난 기억력이 나쁜데.....
- **자동화** 할 수 없을까요??

4.4.4. 자동 미분 with 자동 역전파

- 호출 순서를 역순으로 저장해뒀다가 역전파 계산할때 사용하면 될 것 같아요!

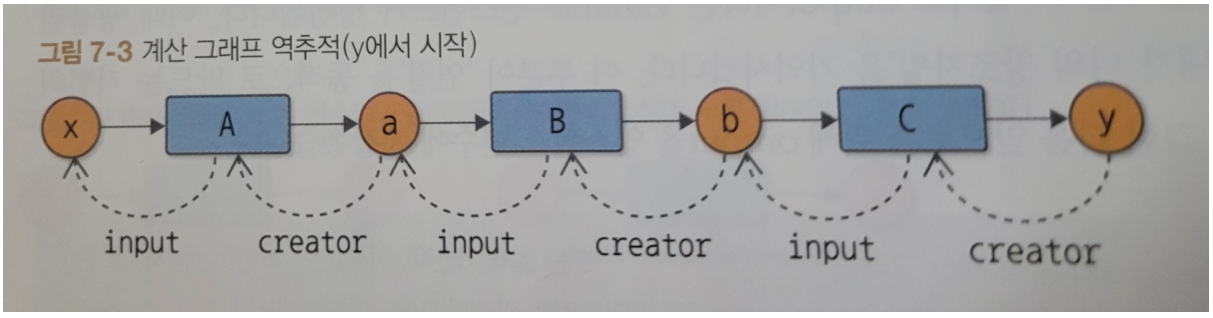
```
A = Square()
B = Exp()
C = Square()

# 순전파 과정
x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b) # y = C(B(A(x)))
order = [C, B, A]

# 역전파 과정
y.grad = np.array(1.0)
for f in order:
    y.grad = f.backward(y.grad)
print(y.grad)

3.297442541400256
```

- 엇 근데, 딥러닝 모델에 사용되는 함수들은 **한 번에 여러 개의 함수와 합성**되는데... 위 방식으로는 이러한 계산을 수행할 수 없을 것 같아요.
    - 어쩌죠??
- 음.. **연산 그래프**를 보면서 조금 고민을 해봐야겠어요.



- 아! 각 **Variable별로 자신을 생성한 함수를 알 수 있으면** 입력의 미분값을 계산할 수 있을 것 같아요!

```
class Variable:
    def __init__(self, data):
        self.data = data
        self.grad = None # 변수에 미분결과 저장
        self.creator = None # 자신을 생성한 Function 객체

    def set_creator(self, func): # 변수의 생성함수 설정 메소드
        self.creator = func

class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(y)
        output.set_creator(self) # 출력 변수의 생성함수를 자신으로 저장
        self.input = input # 미분값 계산시 사용할 값 저장
        self.output = output # 출력 변수 저장
        return output
```

```
A = Square()
B = Exp()
C = Square()

# 순전파 과정
x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b) # y = C(B(A(x)))

# 역전파 과정
y.grad = np.array(1.0)

# 역전파 1
C = y.creator # 생성함수 가져옴
b = C.input # 입력 변수 가져옴
b.grad = C.backward(y.grad) # 입력 변수의 미분값 계산

# 역전파 2
B = b.creator
a = B.input
a.grad = B.backward(b.grad)

# 역전파 3
A = a.creator
x = A.input
x.grad = A.backward(a.grad)
print(x.grad)

3.297442541400256
```

- 위에서 역전파 1,2,3 은 모두 **같은 동작**을 하고 있네요!
  - 이걸 좀 더 단순하게 바꿀 수 있을 것 같아요!

```
class Variable:
    def __init__(self, data):
        self.data = data
        self.grad = None # 변수에 미분결과 저장
        self.creator = None # 자신을 생성한 Function 객체

    def set_creator(self, func): # 변수의 생성함수 설정 메소드
        self.creator = func

    def backward(self): # 재귀적으로 자신을 생성한 함수의
        f = self.creator # 입력변수 미분값을 계산하는 메소드
        if f is not None:
            x = f.input
            x.grad = f.backward(self.grad)
            x.backward()
```

```
A = Square()
B = Exp()
C = Square()

# 순전파 과정
x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b) # y = C(B(A(x)))

# 역전파 과정
y.grad = np.array(1.0)
y.backward()
print(x.grad)

3.297442541400256
```

- **역전파과정까지 자동**으로 계산하도록 만들었어요!
- PyTorch에서처럼 **backward() 호출 한번**으로 모두 계산할 수 있어요!
- 이제 이걸 쓰면 자동미분을 PyTorch한테 요청하지 않아도 되겠죠??
  - 엄마한테 자랑하니까 자동미분 기능만 갖고는 PyTorch를 도와줄 수 없대요.. Model, Optimizer, DataLoader 등등.. PyTorch가 갖고있는 다른 기능들에 이식해 사용하려면 더 많은 준비가 필요하다고 하네요..
  - 아쉽지만.. 실력을 더 키울때 까진 PyTorch의 힘을 빌려야 할 것 같아요.

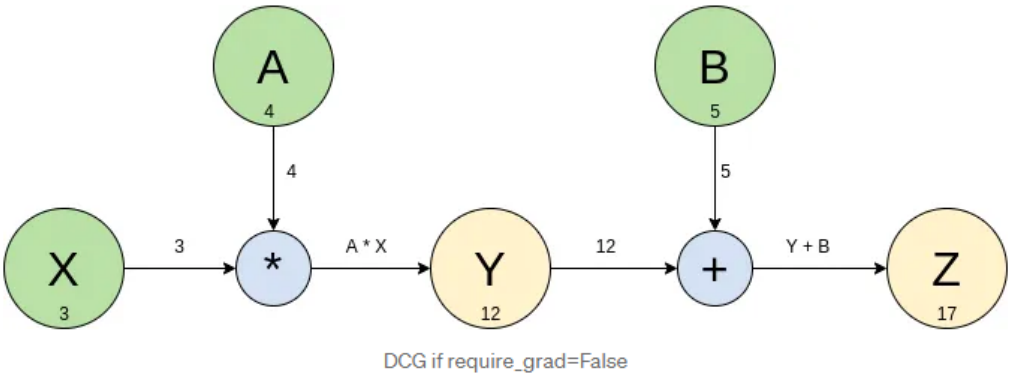
5. Pytorch 의 Autograd [9]

- 위에서 구현한 자동미분과 PyTorch의 Autograd 는 어떻게 다른지 한번 살펴봤어요!

5.1. Autograd

- PyTorch에서는 Variable의 역할을 **Tensor**가 해요.
  - Variable 은 deprecated 되었대요.
    - https://pytorch.org/docs/stable/autograd.html#module-torch.autograd:~:text=gradients are correct.-,Variable (deprecated),-WARNING
    - 예전 소스코드와의 호환성을 위해서 사용할때 내부적으로 Tensor로 바꿔준대요!
      - WARN 메시지도 띄워주고요!
- PyTorch는 **Computation Graph**라는 Map 형태의 자료구조로 **변수와 함수 사이의 관계**를 저장한대요.
  - 우리가 위에서 순전파시에 관계를 저장한 것처럼, PyTorch도 forward pass에 이 Graph를 만든대요!
  - 아래 예시처럼 변수와 함수(=연산자) 사이의 연산 그래프를 생성해요.

```
x = torch.tensor(3.)
a = torch.tensor(4.)
b = torch.tensor(5.)
y = a * x
z = y + b
```



DCG if require_grad=False

  - 근데 모든 Tensor들이 이런 그래프를 **만들지는 않는다네요?**

5.1.1. Tensor의 주요 속성들: requires_grad, grad_fn, is_leaf

- **requires_grad** 는 해당 Tensor가 **자동미분의 대상인지**를 나타내는 표시예요.
  - Autograd가 자동미분을 수행할때, **이 표시가 되어있는 Tensor에 대해서만 미분연산을 수행**한대요.

    - 그리고, Tensor간의 연산에서 하나라도 이 표시를 들고 있으면, 결과 Tensor도 표시를 들게 된대요.
- **grad_fn** 은 우리가 위에서 작성한 코드중 **creator**에 해당하는 친구예요.
    - 해당 Tensor **변수와 연관된 함수(=연산자) 를 참조**하고 있어요.
    - 아 물론, 자동미분의 대상이 아닌 Tensor는 해당 값이 필요없어요!
        - 그러니까 grad_fn 이 None이 되는거죠!
- **is_leaf** 는 해당 Tensor가 Leaf인지 여부를 나타내는 표시예요.
    - Leaf 가 뭐냐면, 우리가 **결과적으로 구하고 싶은 미분값을 갖는 Tensor**를 의미해요!
    - PyTorch는 우리가 구현한 코드와 달리, **backward()시에 Leaf Tensor들만 미분값을 저장**한대요!
    - 자동미분 대상이 아닌 Tensor의 연산으로부터 생성된 Tensor나, 사용자가 직접 생성한 Tensor가 Leaf Tensor 가 된대요!
        - 바로 이전 코드 예제에서는 x, y, z 중 x만 Leaf Tensor인 거죠.(grad_fn이 None이예요!)

5.1.2. backward()

- 이제 강의에서 모든 걸 하나로 설명했던 이 친구에 대해 알아볼게요!
- 이 친구는 현재 Tensor에 연관된 **연산 그래프상에서 Leaf 인 Tensor들의 미분값을 계산**해줘요!
    - 우리는 연관된 **모든 변수**에서 미분값을 계산했는데, **PyTorch는 그렇지 않은가봐요**.
    - 왜 그럴까요??
        - 우리가 **최적화 해야 하는 대상**은 **Leaf Tensor 들이기 때문**이예요!
            - 선형변환 W*x + b 에서 W, b, x는 모두 Leaf Tensor 예요!
        - 필요한 값만 구하면 연산시간과 메모리를 모두 아낄수 있어요!
    - 그럼 Leaf Tensor 가 아닌 Tensor 는 미분값을 가질 수 있는 방법이 없는거예요?
        - 다 같은 Tensor들인데 누구는 사탕을 갖고, 누구는 못 갖고 있으면 불쌍하잖아요..
        - 원하는 친구는 갖고 있게 할 순 없나요??

5.1.3. retain_grad()

- 다행히 원하면 미분값을 갖고 있을 수 있대요!
- retain_grad()를 호출한 텐서는 미분값을 갖게 된대요!
- 이 친구들은 **미분값이 이상할때** 중간에 **도움**을 줄 수 있을것 같아요!

5.1.4. register_hook(grad)

- 이 친구는 Tensor가 본인의 **미분값을 다르게 처리**하도록 도와준대요!
- 갖고있는 미분값은 1인데 돌려줄때 2를 돌려주도록 해주는 마법🪄같은 친구예요!

5.1.5. detach()

- 이 친구는 Tensor 랑 사탕바구니🎁를 공유하는 새로운 Leaf Tensor 를 만들어요.
- 내가 사탕바구니에 딸기사탕🍓을 포도사탕🍇으로 바꿔놓으면, 생성된 텐서 친구도 포도사탕🍇만 먹을 수 있어요.

- 이렇게 PyTorch의 Autograd 에 대해서 살펴봤어요.
    - 아직은 PyTorch의 도움을 많이 받아야할 것 같아서 어떻게 구현되었는지도 한번 찾아봤어요!

5.2. backward() 구현 코드 [10]

```
class Tensor(torch._C.TensorBase):
    def backward(self, gradient=None, retain_graph=None
                         , create_graph=False, inputs=None):
        ... # 중략
        torch.autograd.backward(
                self, gradient, retain_graph, create_graph, inputs=inputs
        )

# torch.autograd
def backward(
    tensors: _TensorOrTensors,
    grad_tensors: Optional[_TensorOrTensors] = None,
    retain_graph: Optional[bool] = None,
    create_graph: bool = False,
    grad_variables: Optional[_TensorOrTensors] = None,
    inputs: Optional[_TensorOrTensorsOrGradEdge] = None,
) -> None:
    r"""Computes the sum of gradients of given tensors with respect to graph
    leaves.

    The graph is differentiated using the chain rule. If any of ``tensors``
    are non-scalar (i.e. their data has more than one element) and require
    gradient, then the Jacobian-vector product would be computed, in this
    case the function additionally requires specifying ``grad_tensors``.
    It should be a sequence of matching length, that contains the "vector"
    in the Jacobian-vector product, usually the gradient of the differentiated
    function w.r.t. corresponding tensors (``None`` is an acceptable value for
    all tensors that don't need gradient tensors).

    This function accumulates gradients in the leaves - you might need to zero
    ``.grad`` attributes or set them to ``None`` before calling it.
    See :ref:`Default gradient layouts<default-grad-layouts>`
    for details on the memory layout of accumulated gradients.

    .. note::
        Using this method with ``create_graph=True`` will create a reference cycle
        between the parameter and its gradient which can cause a memory leak.
        We recommend using ``autograd.grad`` when creating the graph to avoid this.
        If you have to use this function, make sure to reset the ``.grad`` fields of your
        parameters to ``None`` after use to break the cycle and avoid the leak.

    .. note::

        If you run any forward ops, create ``grad_tensors``, and/or call ``backward``
        in a user-specified CUDA stream context, see
        :ref:`Stream semantics of backward passes<bwd-cuda-stream-semantics>`.

    .. note::

        When ``inputs`` are provided and a given input is not a leaf,
        the current implementation will call its grad_fn (even though it is not strictly needed to get this gradients).
        It is an implementation detail on which the user should not rely.
        See https://github.com/pytorch/pytorch/pull/60521#issuecomment-867061780 for more details.

    Args:
        tensors (Sequence[Tensor] or Tensor): Tensors of which the derivative will be
            computed.
        grad_tensors (Sequence[Tensor or None] or Tensor, optional): The "vector" in
            the Jacobian-vector product, usually gradients w.r.t. each element of
            corresponding tensors. None values can be specified for scalar Tensors or
            ones that don't require grad. If a None value would be acceptable for all
            grad_tensors, then this argument is optional.
        retain_graph (bool, optional): If ``False``, the graph used to compute the grad
            will be freed. Note that in nearly all cases setting this option to ``True``
            is not needed and often can be worked around in a much more efficient
            way. Defaults to the value of ``create_graph``.
        create_graph (bool, optional): If ``True``, graph of the derivative will
            be constructed, allowing to compute higher order derivative products.
            Defaults to ``False``.
        inputs (Sequence[Tensor] or Tensor or Sequence[GradientEdge], optional): Inputs w.r.t. which the gradient
            be will accumulated into ``.grad``. All other Tensors will be ignored. If
            not provided, the gradient is accumulated into all the leaf Tensors that
            were used to compute the attr::tensors.
    """
```

```
    ... # 중략

    tensors = (tensors,) if isinstance(tensors, torch.Tensor) else tuple(tensors)
    inputs = (
        (inputs,)
        if isinstance(inputs, (torch.Tensor, graph.GradientEdge))
        else tuple(inputs)
        if inputs is not None
        else tuple()
    )

    grad_tensors_ = _tensor_or_tensors_to_tuple(grad_tensors, len(tensors)) # 튜플형태로 변환
    grad_tensors_ = _make_grads(tensors, grad_tensors_, is_grads_batched=False) # output 과 input 사이즈 비교 등 validation
    if retain_graph is None:
        retain_graph = create_graph

    # The reason we repeat the same comment below is that
    # some Python versions print out the first line of a multi-line function
    # calls in the traceback and some print out the last line
    Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
        tensors,
        grad_tensors_,
        retain_graph,
        create_graph,
        inputs,
        allow_unreachable=True,
        accumulate_grad=True,
    )  # Calls into the C++ engine to run the backward pass

def _tensor_or_tensors_to_tuple(
    tensors: Optional[_TensorOrTensors], length: int
) -> Tuple[_OptionalTensor, ...]:
    if tensors is None:
        return (None,) * length
    if isinstance(tensors, torch.Tensor):
        return (tensors,)
    return tuple(tensors)

def _make_grads(
    outputs: Sequence[torch.Tensor],
    grads: Sequence[_OptionalTensor],
    is_grads_batched: bool,
) -> Tuple[_OptionalTensor, ...]:
    new_grads: List[_OptionalTensor] = []
    for out, grad in zip(outputs, grads):
        if isinstance(grad, torch.Tensor):
            from torch.fx.experimental.symbolic_shapes import expect_true, sym_eq

            first_grad = grad if not is_grads_batched else grad[0]
            # TODO: We can remove this conditional once we uniformly use
            # singleton int to represent jagged dimension, so that size() call
            # on nested tensor works
            if out.is_nested or first_grad.is_nested:
                shape_matches = torch.is_same_size(out, first_grad)
            else:
                # We need to do a regular size check, without going through
                # the operator, to be able to handle unbacked symints
                # (expect_true ensures we can deal with unbacked)
                shape_matches = expect_true(sym_eq(out.size(), first_grad.size()))
            if not shape_matches:
                out_shape, grad_shape = _calculate_shape(
                    out, first_grad, is_grads_batched
                )
                if is_grads_batched:
                    raise RuntimeError(
                        "If `is_grads_batched=True`, we interpret the first "
                        "dimension of each grad_output as the batch dimension. "
                        "The sizes of the remaining dimensions are expected to match "
                        "the shape of corresponding output, but a mismatch "
                        "was detected: grad_output["
                        + str(grads.index(grad))
                        + "] has a shape of "
                        + str(grad_shape)
                        + " and output["
                        + str(outputs.index(out))
```

```
                        + "] has a shape of "
                        + str(out_shape)
                        + ". "
                        "If you only want some tensors in `grad_output` to be considered "
                        "batched, consider using vmap."
                    )
                else:
                    raise RuntimeError(
                        "Mismatch in shape: grad_output["
                        + str(grads.index(grad))
                        + "] has a shape of "
                        + str(grad_shape)
                        + " and output["
                        + str(outputs.index(out))
                        + "] has a shape of "
                        + str(out_shape)
                        + "."
                    )
            if out.dtype.is_complex != grad.dtype.is_complex:
                raise RuntimeError(
                    "For complex Tensors, both grad_output and output"
                    " are required to have the same dtype."
                    " Mismatch in dtype: grad_output["
                    + str(grads.index(grad))
                    + "] has a dtype of "
                    + str(grad.dtype)
                    + " and output["
                    + str(outputs.index(out))
                    + "] has a dtype of "
                    + str(out.dtype)
                    + "."
                )
            new_grads.append(grad)
     ... # 중략
    return tuple(new_grads)


class Variable(torch._C._LegacyVariableBase, metaclass=VariableMeta):  # type: ignore[misc]
    _execution_engine = ImperativeEngine()

# Defined in torch/csrc/autograd/python_engine.cpp
class _ImperativeEngine:
    def queue_callback(self, callback: Callable[[], None]) -> None: ...
    def run_backward(self, *args: Any, **kwargs: Any) -> Tuple[Tensor, ...]: ...
    def is_checkpoint_valid(self) -> _bool: ...
```

```
// Implementation of torch._C._EngineBase.run_backward
PyObject* THPEngine_run_backward(
    PyObject* self,
    PyObject* args,
    PyObject* kwargs) {
  HANDLE_TH_ERRORS
  PyObject* tensors = nullptr;
  PyObject* grad_tensors = nullptr;
  unsigned char keep_graph = 0;
  unsigned char create_graph = 0;
  PyObject* inputs = nullptr;
  unsigned char allow_unreachable = 0;
  unsigned char accumulate_grad =
      0; // Indicate whether to accumulate grad into leaf Tensors or capture
  constexpr const char* accepted_kwargs[] = {// NOLINT
                                             "tensors",
                                             "grad_tensors",
                                             "keep_graph",
                                             "create_graph",
                                             "inputs",
                                             "allow_unreachable",
                                             "accumulate_grad",
                                             nullptr};
  if (!PyArg_ParseTupleAndKeywords(
          args,
          kwargs,
          "OObb|Obb",
```

```
        // NOLINTNEXTLINE(cppcoreguidelines-pro-type-const-cast,-warnings-as-errors)
        const_cast<char**>(accepted_kwargs),
        &tensors,
        &grad_tensors,
        &keep_graph,
        &create_graph,
        &inputs,
        &allow_unreachable,
        &accumulate_grad))
  return nullptr;
... // 중략

edge_list roots;
roots.reserve(num_tensors);
variable_list grads;
grads.reserve(num_tensors);
for (const auto i : c10::irange(num_tensors)) {
  PyObject* _tensor = PyTuple_GET_ITEM(tensors, i);
  THPUtils_assert(
      THPVariable_Check(_tensor),
      "element %d of tensors "
      "tuple is not a Tensor",
      i);
  const auto& variable = THPVariable_Unpack(_tensor);
  TORCH_CHECK(
      !isBatchedTensor(variable),
      "torch.autograd.grad(outputs, inputs, grad_outputs) called inside ",
      "torch.vmap. We do not support the case where any outputs are ",
      "vmapped tensors (output ",
      i,
      " is being vmapped over). Please "
      "call autograd.grad() outside torch.vmap or file a bug report "
      "with your use case.")
  auto gradient_edge = torch::autograd::impl::gradient_edge(variable);
  THPUtils_assert(
      gradient_edge.function,
      "element %d of tensors does not require grad and does not have a grad_fn",
      i);
  roots.push_back(std::move(gradient_edge));

  PyObject* grad = PyTuple_GET_ITEM(grad_tensors, i);
  if (THPVariable_Check(grad)) {
    const Variable& grad_var = THPVariable_Unpack(grad);
    if (grad_var.has_names()) {
      TORCH_WARN(
          "Autograd was passed a named grad tensor with dims ",
          grad_var.names(),
          ". Autograd does not yet support named tensor semantics, so all names ",
          "will be ignored. In practice all computed gradients will still be correct "
          "according to regular tensor semantics.");
    }
    grads.push_back(grad_var);
  } else {
    THPUtils_assert(
        grad == Py_None,
        "element %d of gradients tuple is not a Tensor or None",
        i);
    THPUtils_assert(
        !variable.requires_grad(),
        "element %d of gradients tuple is None, but the corresponding Tensor requires grad",
        i);
  }
}

std::vector<Edge> output_edges;
if (inputs != nullptr) {
  TORCH_CHECK(
      PyTuple_CheckExact(inputs), "inputs to run_backward must be a tuple");
  int num_inputs = PyTuple_GET_SIZE(inputs);
  output_edges.reserve(num_inputs);
  for (const auto i : c10::irange(num_inputs)) {
    PyObject* input = PyTuple_GET_ITEM(inputs, i);
    if (THPVariable_Check(input)) {
      const auto& tensor = THPVariable_Unpack(input);
      TORCH_CHECK(
```

```
            !isBatchedTensor(tensor),
            "torch.autograd.grad(outputs, inputs, grad_outputs) called inside ",
            "torch.vmap. We do not support the case where any inputs are ",
            "vmapped tensors (input ",
            i,
            " is being vmapped over). Please "
            "call autograd.grad() outside torch.vmap or file a bug report "
            "with your use case.")
        const auto output_nr = tensor.output_nr();
        auto grad_fn = tensor.grad_fn();
        if (!grad_fn) {
          grad_fn = torch::autograd::impl::try_get_grad_accumulator(tensor);
        }
        if (accumulate_grad) {
          tensor.retain_grad();
        }
        THPUtils_assert(
            tensor.requires_grad(),
            "One of the differentiated Tensors does not require grad");
        if (!grad_fn) {
          // NOTE [ Autograd Unreachable Input ]
          // Since input has no grad_accumulator, its guaranteed to be
          // unreachable. We initialize an edge pointing to a non-nullptr Node
          // so nodes in the graph (e.g., mul when an operand is scalar) that
          // have edges pointing to nullptr don't get erroneously assigned
          // `needed = True` in exec_info.
          output_edges.emplace_back(std::make_shared<Identity>(), 0);
        } else {
          output_edges.emplace_back(grad_fn, output_nr);
        }
      } else if (PyObject_IsInstance(input, THPGradientEdgeClass)) {
        auto node = PyTuple_GetItem(input, 0);
        bool isTHPFunction = THPFunction_Check(node);
        bool isTHPCppFunction = THPCppFunction_Check(node);
        THPUtils_assert(
            isTHPFunction || isTHPCppFunction,
            "GradientEdge first object must be an autograd.graph.Node "
            "but got %s",
            THPUtils_typename(node));
        std::shared_ptr<torch::autograd::Node> node_sp;
        if (isTHPFunction) {
          node_sp = ((THPFunction*)node)->cdata.lock();
        } else {
          node_sp = ((torch::autograd::THPCppFunction*)node)->cdata;
        }

        auto output_nr = THPUtils_unpackUInt32(PyTuple_GetItem(input, 1));
        output_edges.emplace_back(node_sp, output_nr);
      } else {
        THPUtils_assert(
            false,
            "all inputs have to be Tensors or GradientEdges, but got %s",
            THPUtils_typename(input));
      }
    }
  }

  variable_list outputs; // 결과
  {
    pybind11::gil_scoped_release no_gil; // gil 해제, 파이썬 코드를 실행하기 위함
    auto& engine = python::PythonEngine::get_python_engine(); // 파이썬 실행 엔진 가져옴
    outputs = engine.execute( // roots부터 파이썬 코드실행
        roots, grads, keep_graph, create_graph, accumulate_grad, output_edges);
  }
  ... // 중략

}

variable_list PythonEngine::execute(
    const edge_list& roots,
    const variable_list& inputs,
    bool keep_graph,
    bool create_graph,
    bool accumulate_grad,
    const edge_list& outputs) {
```

```
  TORCH_CHECK(
      !PyGILState_Check(),
      "The autograd engine was called while holding the GIL. If you are using the C++ "
      "API, the autograd engine is an expensive operation that does not require the "
      "GIL to be held so you should release it with 'pybind11::gil_scoped_release no_gil;'"
      ". If you are not using the C++ API, please report a bug to the pytorch team.")
  try {
    return Engine::execute(
        roots, inputs, keep_graph, create_graph, accumulate_grad, outputs);
  } catch (python_error& e) {
    e.restore();
    throw;
  }
```

```
auto Engine::execute(
    const edge_list& root_edges,
    const variable_list& inputs,
    bool keep_graph,
    bool create_graph,
    bool accumulate_grad,
    const edge_list& outputs) -> variable_list {
  validate_outputs(
      root_edges,
      // NOLINTNEXTLINE(cppcoreguidelines-pro-type-const-cast)
      const_cast<variable_list&>(inputs),
      [](const std::string& msg) { return msg; });
  if (accumulate_grad && create_graph) {
    TORCH_WARN_ONCE(
        "Using backward() with create_graph=True will create a reference cycle "
        "between the parameter and its gradient which can cause a memory leak. "
        "We recommend using autograd.grad when creating the graph to avoid this. "
        "If you have to use this function, make sure to reset the .grad fields of "
        "your parameters to None after use to break the cycle and avoid the leak.");
  }

  // Allows us to assert no other threads are in backwards
  CompiledAutogradThreadingDebugCheck _thread_check;
  auto compiled_autograd = the_compiled_autograd.load();
  TORCH_INTERNAL_ASSERT(compiled_autograd != COMPILED_AUTOGRAD_POISON);

  // accumulate_grad is true if and only if the frontend call was to
  // grad(), not backward(). grad() returns the sum of the gradients
  // w.r.t. the inputs and thus needs the inputs to be present.
  TORCH_CHECK_VALUE(
      accumulate_grad || !outputs.empty(), "grad requires non-empty inputs.");

  // A fresh first time Engine::execute call should start on the CPU device,
  // initialize a new thread local ready queue on CPU or reuse the existing one
  // (if there is one allocated already, i.e. consecutive backward calls,
  // re-entrant backward calls), then memoize the local_ready_queue in GraphTask
  init_local_ready_queue();
  bool not_reentrant_backward_call = worker_device == NO_DEVICE;

  // Store root nodes so we can traverse through the graph later
  // e.g., for get_current_graph_task_execution_order
  c10::SmallVector<Node*, 4> temp_roots{root_edges.size()};
  for (const auto i : c10::irange(root_edges.size())) {
    temp_roots[i] = root_edges[i].function.get();
  }

  auto graph_task = std::make_shared<GraphTask>(
      /* keep_graph */ keep_graph,
      /* create_graph */ create_graph,
      /* depth */ not_reentrant_backward_call ? 0 : total_depth + 1,
      /* cpu_ready_queue */ local_ready_queue,
      /* graph_roots */ std::move(temp_roots));

  // If we receive a single root, skip creating extra root node
  bool skip_dummy_node = root_edges.size() == 1 && compiled_autograd == nullptr;
  auto graph_root = skip_dummy_node
      ? root_edges.at(0).function
      : std::make_shared<GraphRoot>(root_edges, inputs);
```

```
  auto min_topo_nr = compute_min_topological_nr(outputs);
  // Now compute the dependencies for all executable functions
  compute_dependencies(graph_root.get(), *graph_task, min_topo_nr);

  if (!outputs.empty()) {
    graph_task->init_to_execute(
        *graph_root, outputs, accumulate_grad, min_topo_nr);
  }

  if (compiled_autograd != nullptr) {
    // see [Note: Compiled Autograd]
    TORCH_CHECK(
        !create_graph, "compiled_autograd does not support create_graph");
    _thread_check.release();
    TORCH_CHECK(
        !AnomalyMode::is_enabled(),
        "compiled_autograd does not support AnomalyMode")
    return (*compiled_autograd)(
        graph_root, *graph_task, accumulate_grad, outputs);
  }

  // Queue the root
  if (skip_dummy_node) {
    InputBuffer input_buffer(root_edges.at(0).function->num_inputs());
    auto input = inputs.at(0);

    const auto input_stream = InputMetadata(input).stream();
    const auto opt_next_stream =
        root_edges.at(0).function->stream(c10::DeviceType::CUDA);
    input_buffer.add(
        root_edges.at(0).input_nr,
        std::move(input),
        input_stream,
        opt_next_stream);

    execute_with_graph_task(
        graph_task, std::move(graph_root), std::move(input_buffer));
  } else {
    execute_with_graph_task(
        graph_task, std::move(graph_root), InputBuffer(variable_list()));
  }
  // Avoid a refcount bump for the Future, since we check for refcount in
  // DistEngine (see TORCH_INTERNAL_ASSERT(futureGrads.use_count() == 1)
  // in dist_engine.cpp).
  auto& fut = graph_task->future_result_;
  fut->wait();
  graph_task->warning_handler_.replay_warnings();
  return fut->value().toTensorVector();
}
```

6. 결론

- PyTorch는 **프레임워크**로서 **딥러닝 모델 학습 전반에 필요한 다양한 기능들을 부탁받는 방식대로 처리해주는 친구**예요.
- PyTorch가 받아주는 부탁중엔 **자동미분** 이라는게 있는데, PyTorch는 이걸 **Autograd**라고 불러요.
- 자동미분은 **컴퓨터를 사용해 미분값을 구하는 것**을 의미하고, 조금 더 엄밀한 의미론 **미분의 연쇄법칙을 사용하는 방식**을 의미해요.
- 자동미분은 순전파시 **호출된 함수의 순서를 외우지 않아도 돼서** 편해요.
- 순전파시 생성된 **변수와 함수의 연관관계를 저장**해두고, 최종 결과 변수로부터 생성 함수가 없을때까지(입력변수까지) 미분값 계산을 전파시켜요.

7. Reference

- [1] (2강) PyTorch Basics
- [2] https://aws.amazon.com/ko/what-is/framework/

- [3] https://ddungsang.tistory.com/602
- [4] https://chat.openai.com/c/9a6c207b-27d5-40dd-97fe-f37a99fc3350#:~:text=PyTorch는 딥러닝 모델을 만들고 학습시키기 위한 고수준의 추상화와 제어의 흐름을 가지고 있습니다. 모델의 구조%2C 학습 알고리즘%2C 데이터 로딩 등의 부분에서 프레임워크가 개발자에게 제어의 주도권을 주고 있습니다. 따라서 PyTorch는 프레임워크로 분류됩니다.
- [5] https://instablank.com/jjalbangMake/10159
- [6] https://everymemes.tistory.com/202
- [7] 밑바닥부터 시작하는 딥러닝3_사이토 고키 https://www.yes24.com/Product/Goods/95343845
- [8] https://blog.naver.com/mykepzzang/220072089756
- [9] https://medium.com/@namanphy/understanding-autograd-5-pytorch-tensor-functions-8f47c27dc38
- [10] https://github.com/pytorch/pytorch/blob/main/torch/csrc/autograd/engine.cpp#L1163