Implementation of Computation Graph

딥러닝을 목적으로 사용하는 계산 그래프를 어떻게 구현하는지에 대한 고찰이다. 단순히 층을 쌓는 방식의 구조는 설계도 구현도 쉽지만, 신경망의 아키텍쳐가 조금만 복잡해져도 대응이 불가능할 수 있다. (ex: U-Net) 때문에 `TensorFlow` 나 `PyTorch` 등의 프레임워크는 모두 계산 그래프 모델을 사용한다.

이 글에서는 글쓴이가 만들고 있는 구조를 검증하고자 쓰였다.

Definition

`Flux` : `numpy.ndarray` 를 매체로 삼는 x와 $\dfrac{\partial L}{\partial x}$를 저장하는 객체

`FluxOp` : 0개 이상의 `Flux` 로부터 1개 이상의 `Flux` 를 계산하는 객체

`FluxNet` : 1개 이상의 `FluxOp` 로 이루어진 객체로, `FluxNet.forward` 와 `FluxNet.backward` 를 갖는다.

Computation Priority Problem

`FluxNet` 은 반드시 올바른 계산 순서로 `FluxOp` 를 작동시켜야 한다.

올바른 계산 순서 $\mathrm{order}(x)$는 `FluxNet.forward` 를 올바르게 계산하는 순서로서, 다음과 같이 정의한다.

1. 독립된 `Flux` x에 대하여 $\mathrm{order}(x)$는 임의로 음이 아닌 정수로 정의할 수 있다.
2. $f \in \mathrm{FluxOp}, x_1, x_2, \ldots \in \mathrm{Flux}$이면 $\mathrm{order}(f(x_1, x_2, \ldots)) > \max(\mathrm{order}(x_1), \mathrm{order}(x_2), \ldots)$를 만족한다.

이를 만족시키려면 다음과 같은 구조를 생각할 수 있다. (실제로 과거에 무식했던 글쓴이가 Java로 구현했던 것을 Python으로 바꾼 것이다)

```python
import numpy as np

# 이 코드는 비효율적인 방식을 사용하므로, 실전에는 사용하지 말 것
class Flux:
    def __init__(self, init, order=0):
        self.order = order
        self.x = init
        self.dx = np.zeros_like(init)

class FluxOpAdd:
    def __init__(self, in_list, out_list):
        self.in_list = in_list
        self.out_list = out_list

    def forward(self):
        # out_list[0]이 2개 이상의 ops로부터 그라디언트를 전달받을 수 있으므로
        # forward를 수행할 때 0으로 초기화하고 이후 backward를 수행할 때 더해준다.
        self.out_list[0].x = self.in_list[0].x + self.in_list[1].x
```

```python
        self.out_list[0].dx.fill(0)

    def backward(self):
        # 더해주는 이유는 forward 주석 참고
        self.in_list[0].dx += self.out_list[0].dx
        self.in_list[1].dx += self.out_list[0].dx

class FluxNet:
    def __init__(self):
        # ops는 FluxOp[][]로, FluxOp의 입력 Flux가 가진 최대 order에 들어간다.
        # Flux의 order가 동일할 수도 있으므로 각 차수별로 또 리스트를 가진다.
        self.ops = []

    def __addFluxOp(self, op, order):
        # 만약 현재 가지고 있는 차수보다 더 높은 order가 나타나면
        # 그 차수를 저장할 수 있도록 공간을 늘린다.
        while len(self.ops) <= order:
            self.ops.append([])
        self.ops[order].append(op)

    def Add(self, flux_a, flux_b):
        # 올바른 계산 순서 조건을 만족시키기 위함
        max_order = max(flux_a.order, flux_b.order)
        flux_c = Flux(np.zeros_like(flux_a), max_order + 1)
        self.__addFluxOp(FluxOpAdd([flux_a, flux_b], [flux_c]), max_order)
        return flux_c

    def forward(self):
        # 같은 order끼리의 계산 순서는 정의되지 않았으므로 임의로 반복한다.
        for op_list in self.ops:
            for opr in op_list:
                opr.forward()

    def backward(self):
        for op_list in self.ops[::-1]:
            for opr in op_list:
                opr.backward()

# 테스트 코드
flux_net = FluxNet()
flux_x = Flux(np.array([[1, 2], [3, 4]]))
flux_y = Flux(np.array([[5, 6], [7, 8]]))
flux_z = flux_net.Add(flux_x, flux_y)
flux_net.forward()
print(flux_z.x)
# [[ 6  8]
#  [10 12]]

flux_z.dx = np.array([[1, 0], [0, 1]])
flux_net.backward()
print(flux_x.dx)
print(flux_y.dx)
# [[1 0
```

```
#   0 1]]
# [[1 0
#   0 1]]
```

위 코드는 아주 잘 작동한다. 하지만 `FluxNet.Add()` 에서 `Flux.order` 를 건드린다는 건 좀 지저분해보인다. `FluxNet` 이 `FluxOp` 를 저장하는 방식도 영 마음에 안든다. 글쓴이는 다른 방법을 생각해보기도 했다. 하지만 임의의 그래프 구조를 표현하면서, 순서에 맞게 외부에서 그래프를 순회하는 것은 매우 어려웠다.

하지만 조금 더 생각해보자, 글쓴이는 "`Flux` 를 만든 순서가 곧 *올바른 계산 순서*"라는 황당한 결론을 내리게 되었다. 왜 그런지 증명해보자.

Proof

`FluxOp` 의 나열 (s_n)이 있다고 가정하자. 문제를 단순화하기 위해 모든 `FluxOp` 는 1개의 출력만을 갖는다. 2개 이상의 출력에 대해서는, 같은 입력을 갖는 서로 다른 가상의 `FluxOp` 가 존재한다고 해석하면 된다. (s_n)을 형성하면서 `Flux` 의 나열 (t_n)이 생성된다. 단 최초의 t_1을 만드는 과정에는 `Flux` (u_m)이 순서대로 사용되었으며 나머지 t_n은 임의의 u_i나 t_j $(\text{단}, j < n)$을 사용하여 만들었다.

증명하고자 하는 명제에 따라, 각각의 *올바른 계산 순서*를 아래와 같이 생성 순서대로 정한다.

$$\begin{aligned}\text{order}(u_k) &= k \\ \text{order}(t_k) &= m + k\end{aligned}$$

증명을 하기 전에 편의를 위한 정의와 보조정리 하나를 증명한다.

Definition:

$$\begin{gathered} m = \max(X) \iff m \in X \land \forall_{x \in X}(x \le m) \\ \text{order}(X) \equiv \{\text{order}(x) | x \in X\} \\ \text{order}(\emptyset) \equiv 0 \end{gathered}$$

Lemma:

$$A \subseteq B \implies \max(A) \le \max(B)$$

Proof

$$\begin{gathered} \max(A) \in A \land A \subseteq B \implies \max(A) \in B \\ \max(A) \in B \land \forall_{b \in B}(b \le \max(B)) \implies \max(A) \le \max(B) \end{gathered}$$

이제 모든 `Flux` 에 대하여 정의한 order이 *올바른 계산 순서*의 조건을 만족하는지 보이면 된다.

(u_m)은 독립된 `Flux` 이므로 조건 1에 의해 자명하다.

(t_n)는 다음과 같이 증명한다. 먼저 집합을 하나 정의한다.

$$\begin{aligned} U_{s_k} &= \{u_\sigma | u_\sigma \text{ used for creation of } s_k\} \\ T_{s_k} &= \{t_\sigma | t_\sigma \text{ used for creation of } s_k\} \end{aligned}$$

그러면

$$U_{s_n} \subseteq \{u_1, u_2, \ldots, u_m\} \implies \max(\text{order}(U_{s_n})) \leq \max(\text{order}(u_1), \text{order}(u_2), \ldots) = m$$
$$T_{s_n} \subseteq \{t_1, t_2, \ldots, t_{n-1}\} \implies \max(\text{order}(T_{s_n})) \leq \max(\text{order}(t_1), \text{order}(t_2), \ldots) = m+n-1$$
$$\text{order}(t_n) = m+n > m+n-1 > \max(\text{order}(U_{s_n}), \text{order}(T_{s_n}))$$

따라서 조건 2를 만족한다.

Improved Implementation

위의 결론에 따라 더이상 `Flux` 개체들은 우선순위를 저장할 필요가 없으며, `FluxNet`도 단순히 `FluxOp` 개체들을 생기는 순서대로 저장하기만 하면 된다. 개선된 구현은 다음과 같다.

```
import numpy as np

class Flux:
    def __init__(self, init):
        self.x = init
        self.dx = np.zeros_like(init)

class FluxOpAdd:
    def __init__(self, in_list, out_list):
        self.in_list = in_list
        self.out_list = out_list

    def forward(self):
        # out_list[0]이 2개 이상의 ops로부터 그라디언트를 전달받을 수 있으므로
        # forward를 수행할 때 0으로 초기화하고 이후 backward를 수행할 때 더해준다.
        self.out_list[0].x = self.in_list[0].x + self.in_list[1].x
        self.out_list[0].dx.fill(0)

    def backward(self):
        # 더해주는 이유는 forward 주석 참고
        self.in_list[0].dx += self.out_list[0].dx
        self.in_list[1].dx += self.out_list[0].dx

class FluxNet:
    def __init__(self):
        self.ops = []

    def __addFluxOp(self, op):
        self.ops.append(op)

    def Add(self, flux_a, flux_b):
        flux_c = Flux(np.zeros_like(flux_a))
        self.__addFluxOp(FluxOpAdd([flux_a, flux_b], [flux_c]))
        return flux_c

    def forward(self):
        for opr in self.ops:
            opr.forward()

    def backward(self):
```

```
        for opr in self.ops[::-1]:
            opr.backward()
```

전보다 훨씬 코드가 간결해진 것을 볼 수 있다. 불필요한 2중 for문도 단일 루프로 줄였다.