

Report

김준호

1번 과제: 자동미분엔진 이해하기

I. tensor

1. **Tensor**, **TensorNode** 의 각 attribute이 가지는 의미 (어떻게 사용되는지)

(1) **TensorNode.arr**: 입력 데이터(float, ndarray, 또는 TensorNode)를 ndarray로 변환하여 저장.

(2) **TensorNode.requires_grad**: 역전파 시 기울기를 계산할지의 여부.

(3) **TensorNode.is_leaf**: **requires_grad** 와의 관계:

TensorNode는 (**is_leaf**, **requires_grad**)의 모든 조합을 가질 수 있다는 것을 알 수 있다. 이 조합은 **TensorNode**가 생성될 때의 컨텍스트와 그 역할에 따라 달라진다.

is_leaf는 해당 **TensorNode**가 다른 노드로부터 직접적으로 생성되었는지, 즉 계산 그래프에서 리프 노드인지 여부를 나타낸다. 만약 **TensorNode**가 다른 노드의 연산 결과로 생성되지 않고 직접 초기화되었다면, **is_leaf**는 **True**가 된다. **requires_grad** 속성은 해당 노드의 텐서 값에 대한 기울기가 필요한지, 즉 해당 노드가 기울기 계산 과정에 포함되어야 하는지를 나타낸다.

---(**is_leaf**, **requires_grad**)의 가능한 조합은 2*2로 네 가지일까?

(**True**, **True**), (**True**, **False**), (**False**, **True**), (**False**, **False**) 네 가지가 나온다.

(4) **TensorNode.grad_fn**: 노드를 생성한 연산을 나타내는 함수. 미분에 사용됨.

(5) **TensorNode.grad**: 노드의 기울기(ndarray)를 저장. 초기값은 **None**.

(6) **TensorNode.grad_cnt**: 노드의 기울기 계산에 필요한 연산의 수를 카운트.

(7) **TensorNode.backward**: 역전파를 시작하는 역할을 함. **grad_fn**이 정의되어 있어야 하며, 스칼라 값인 경우에만 호출 가능. 기울기를 1로 초기화하고,

grad_fn을 통해 역전파를 수행.

(8) `TensorNode._create_new_tensornode` 모든 줄을 한 줄씩 해석

```
def _create_new_tensornode(  
    self,  
    o: NodeValue,  
    operation: _NPOperation,  
    grad_fn: Type[GradFn]  
) -> TensorNode:
```

: 이 부분은 줄로 나누기 애매하기 때문에 한 번에 변수들을 설명하겠다. `_create_new_tensornode` 메서드를 정의하고 있다. 이 메서드는 새로운 `TensorNode` 객체를 생성하는 기능을 한다. `self`는 현재 객체의 인스턴스를 가리킨다. `o`는 연산에 사용될 또 다른 값으로, `NodeValue` 타입을 가진다. 이는 `float`나 `TensorNode`일 수 있다. `operation`은 `_NPOperation` 타입으로, 두 `ndarray` 객체를 입력받아 새로운 `ndarray`를 반환하는 함수이다. 이는 두 `TensorNode` 간의 연산을 정의한다. `grad_fn`은 `GradFn` 타입으로, 주어진 연산에 대한 기울기 계산 함수를 정의하는 클래스이다. 이 메서드는 `TensorNode` 타입의 객체를 반환한다.

```
if not isinstance(o, TensorNode):
```

```
    o = TensorNode(o)
```

: `o`가 `TensorNode` 인스턴스가 아니면, `o`를 `TensorNode`의 인스턴스로 변환한다. 이는 모든 연산이 `TensorNode` 객체들 사이에서 이루어지도록 보장한다.

```
new_arr = operation(self.arr, o.arr)
```

: `self.arr`와 `o.arr`에 대해 주어진 `operation`을 수행하여 `new_arr`에 저장한다.

```
if self.requires_grad or o.requires_grad:
```

`self` 또는 `o` 중 하나라도 기울기 계산이 필요한 경우 (`requires_grad=True`), 다음 블록의 코드를 실행한다.

```
    new_requires_grad = True
```

```
new_is_leaf = False
new_grad_fn = grad_fn(self, o)
```

새로운 TensorNode 객체는 기울기 계산이 필요함을 의미하는 new_requires_grad=True로 설정한다. new_is_leaf=False로 설정하여, 새로운 노드가 연산의 결과로 생성된 비리프 노드임을 나타낸다. grad_fn(self, o)를 호출하여 새롭게 생성된 노드에 대한 기울기 함수 객체를 생성하고 new_grad_fn에 저장한다.

```
if self.requires_grad:
    self.grad_cnt += 1
if o.requires_grad:
    o.grad_cnt += 1
```

: self가 기울기 계산이 필요하다면, self.grad_cnt를 1 증가시킨다. 이는 역전파 과정에서 이 노드의 기울기를 얼마나 계산해야 하는지를 추적하기 위함이다. o도 기울기 계산이 필요하다면, o.grad_cnt를 1 증가시킨다. 이 과정은 두 노드의 기울기 누적에 관한 정보다.

```
else:
    new_requires_grad = False
    new_is_leaf = True
    new_grad_fn = None
```

: self와 o 모두 기울기 계산이 필요하지 않은 경우를 처리한다. 새로운 TensorNode 객체는 기울기 계산이 필요하지 않으므로 new_requires_grad=False로 설정한다. 이 노드는 연산의 결과로 생성되지 않은 리프 노드이므로 new_is_leaf=True로 설정한다. 기울기 계산이 필요 없으므로 new_grad_fn은 None으로 설정한다.

```
new_TensorNode = TensorNode(
    arr=new_arr,
    requires_grad=new_requires_grad,
    is_leaf=new_is_leaf,
```

```
grad_fn=new_grad_fn  
)
```

: 이 부분도 줄 단위로 나누기엔 애매하다. 앞에서 설정된 값을 사용하여 새로운 TensorNode 객체를 생성한다. arr에는 연산 결과인 new_arr를, requires_grad에는 new_requires_grad를, is_leaf에는 new_is_leaf를, grad_fn에는 new_grad_fn을 전달하여 초기화한다.

```
return new_TensorNode
```

: 생성된 새로운 TensorNode를 반환한다.

(9) **TensorNode._operation**: 새로운 연산을 생성하는 데 사용된다. 주어진 grad_fn과 operation에 기반하여 TensorNode 객체 간의 연산을 수행할 수 있도록 new_operation 함수를 정의하고 반환한다.

(10) **TensorNode._assert_not_leaf**: requires_grad가 True인 경우, 리프 노드가 아님을 보장하는 데 사용된다. 이 데코레이터는 메서드 호출 시 해당 조건을 확인한다.

(11) **Tensor.__setitem__**: Tensor 객체에서 인덱스를 사용하여 값을 설정하는 메서드이다. requires_grad가 True인 경우, 리프 노드가 아니어야 하며, 새로운 TensorNode를 생성하여 grad_fn을 적절히 설정한다. value가 Tensor일 경우, 배열 값을 설정하고 grad_fn을 SetitemTensorGradFn 또는 SetitemGradFn으로 설정한다.

---왜 TensorNode가 아닌 Tensor에 구현되어 있는가?

역할 분담이다. Tensor 클래스는 사용자 인터페이스를 제공하는 역할을 하며, 고수준의 API를 통해 쉽게 데이터 조작이 가능하다. TensorNode는 실제 데이터를 포함하고 계산 그래프 내에서 동작한다. 또 사용자 편의를 위해서 그렇게 되어 있다. __setitem__은 인덱스를 사용하여 값을 설정하는 일반적인 기능으로, 사용자가 Tensor 객체를 직접 다룰 때 자연스러운 인터페이스를 제공한다. TensorNode가 아닌 Tensor에서 이를 구현함으로써, 복잡한 계산 그래프의 내부 구조를 숨기고 단순한 API를 제공한다.

2. 그래서 **Tensor** 와 **TensorNode** 는 각각 무엇인가? 왜 분리되어 있는가?

Tensor는 사용자 인터페이스를 제공하는 고수준의 API 객체이다. 계산 그래프와 관련된 복잡한 내부 구현을 숨기고, 사용자가 쉽게 사용할 수 있도록 다양한 연산자 오버로딩과 메서드를 제공한다. **TensorNode**를 내부적으로 래핑하여 동작한다. **TensorNode**는 계산 그래프의 구성 요소로, 실제 데이터와 관련 연산을 포함한다. 각 **TensorNode**는 계산 그래프 내에서의 위치와 역할을 나타내며, 연산의 결과로 생성되고, 기울기 계산에 필요한 정보를 저장한다.

왜 분리되어 있는가?

두 클래스를 분리함으로써, 각 클래스는 자신만의 책임을 갖게 된다. **TensorNode**는 계산 그래프와 관련된 저수준의 작업을 담당하고, **Tensor**는 사용자 친화적인 API를 제공한다. 이는 코드의 모듈성과 유지보수성을 높인다. 유연성 때문도 있다. **Tensor**는 다양한 기능을 제공하며, 새로운 연산을 추가하거나 인터페이스를 개선할 때 쉽게 변경할 수 있다. 반면, **TensorNode**는 계산 그래프와 관련된 핵심 기능을 안정적으로 유지할 수 있다. 또한 복잡한 계산 그래프의 구현 세부 사항을 **TensorNode**로 감추고, 사용자는 **Tensor**를 통해 직관적이고 간단하게 데이터를 조작할 수 있다. 이는 사용자 경험을 향상시킨다.

3. Python 문법

(1) **Kwargs**: `**kwargs` (keyword arguments)는 함수에 이름이 지정된 변수 개수가 변할 수 있는 인수를 전달할 때 사용된다. 이를 통해 함수 호출 시 아무 개수나 원하는 키워드 인수를 넘겨줄 수 있다. `**kwargs`는 키워드 인수를 딕셔너리 형태로 받아들이며, 함수 내부에서 `kwargs['키워드']` 형식으로 접근하여 사용할 수 있다.

(2) **self**는 Python 키워드일까?: `self`는 파이썬에서 클래스의 메서드가 자기 자신의 인스턴스를 참조할 수 있도록 해주는 기본적인 인자이다. `self`는 공식적인 내장 Python 키워드는 아니고 객체지향 프로그래밍을 위해 관례적으로 사용되는 첫 번째 매개변수로, 메서드를 호출할 때 파이썬 인터프리터가 자동으로 해당 객체를 전달한다.

(3) **property**, **property.setter**: `getter`와 `setter`의 개념 (어느 언어에서든)

getter는 클래스의 특정 속성 값을 검색하는 메서드이다. 속성의 값을 반환한다. **setter**는 클래스의 특정 속성에 값을 설정하는 메서드. 속성의 값을 설정하고, 필요한 검증 또는 처리를 수행할 수 있다.

Python에서는 `@property`를 사용하여 `getter`를, `@property.setter`를 사용하여 `setter`를 정의할 수 있다.

`property`는 클래스의 메서드를 속성처럼 사용할 수 있게 해주는 데코레이터다. `property` 데코레이터를 사용하면 클래스의 메서드를 호출할 때 메서드 이름 뒤에 괄호를 붙이지 않고도 호출할 수 있게 된다. `property.setter` 데코레이터는 속성에 값을 할당할 때 사용되는 세터 메서드를 정의할 때 사용된다. 세터 메서드를 통해 값을 설정하는 로직을 커스텀할 수 있다.

class property 의 작동 원리에 대해 자세히 설명하세요!

`class property`는 파이썬에서 속성을 정의하고 접근하는데 사용하는 방식 중 하나이다. 클래스의 속성을 간단하게 정의하고, 그 속성에 대한 접근, 수정, 삭제를 제어할 수 있다. `property`는 주로 `getter`, `setter`, `deleter` 메서드와 함께 사용되며, 이 메서드들은 각각 속성을 가져오고, 설정하고, 삭제하는 동작을 정의한다.

`property`는 파이썬의 내장 함수로, 클래스 속성을 정의할 때 `getter`, `setter`, `deleter` 메서드를 함께 정의하여 속성에 대한 접근을 제어할 수 있다. `getter`, `setter`, `deleter`는 각각 속성을 가져오거나(`get`), 설정하거나(`set`), 삭제할 때(`delete`) 호출되는 메서드이다. `property` 데코레이터를 사용하여 이 메서드들을 지정할 수 있다. 이를 통해 속성에 직접 접근하는 것처럼 보이지만 실제로는 메서드를 통해 접근하게 된다.

getter, setter, deleter 메서드는 무엇을 받아서 무엇을 반환하는가?

getter 메서드는 아무 인자도 받지 않고, 속성의 현재 값을 반환한다. 예를 들어, `@property` 데코레이터가 붙은 메서드는 해당 속성을 호출할 때 실행되며, 그 결과를 반환한다.

setter 메서드는 하나의 인자를 받으며, 이 인자는 설정하려는 새로운 값이다. 이 메서드는 속성의 값을 변경하며, 일반적으로 반환값은 없다. `@속성명.setter` 데코레이터로 지정된다.

deleter 메서드는 아무 인자도 받지 않으며, 속성을 삭제하는 동작을 수행한다. 이 메서드는 속성을 삭제할 때 호출되며, 반환값은 없다. `@속성명.deleter`

데코레이터로 지정된다.

(4) typing

TypeVar, ParamSpec, Concatenate에 대해 자세히 설명하세요!

:TypeVar, ParamSpec, Concatenate의 사용은 파이썬의 타이핑 모듈에서 제네릭 타입이나 파라미터의 타입을 정의할 때 사용된다.

TypeVar: 제네릭 타입을 정의할 때 사용되는 도구이다. 제네릭은 함수나 클래스가 여러 데이터 타입에 대해 동작할 수 있도록 한다. TypeVar를 사용하면 특정 타입의 변수에 대해 보다 구체적인 타입 제한을 설정할 수 있다.

ParamSpec: 파이썬 3.10부터 도입된 기능으로, 함수의 매개변수를 정의하고 추적하는데 사용된다. 이는 함수의 매개변수를 다른 함수로 전달하거나, 데코레이터와 같은 고차 함수에서 매개변수를 유지할 때 유용하다.

Concatenate: ParamSpec과 함께 사용되어 함수의 매개변수 목록을 확장할 때 사용된다. 이는 함수의 앞쪽이나 뒤쪽에 추가적인 매개변수를 삽입해야 할 때 유용하다.

제네릭의 개념, 언제 쓰이는가? 어떻게 쓰이는가?

제네릭은 데이터 타입을 generalize함으로써, 다양한 데이터 타입에서 동작할 수 있는 함수, 클래스, 인터페이스 등을 생성할 수 있게 하는 프로그래밍 기법이다. 코드의 재사용성을 높이고, 타입 안정성을 보장할 수 있을 때 사용한다. 예를 들어, 여러 타입의 리스트를 처리하는 함수를 하나만 정의하여 각기 다른 타입에 대해 사용할 수 있도록 할 때 유용하다. 파이썬에서는 TypeVar을 사용하여 제네릭 타입을 정의하고, 이 타입 변수를 함수나 클래스 정의에 사용함으로써 제네릭 프로그래밍을 수행한다.

II. autograd

GradFn의 작동 원리

1. class GradFn의 모든 줄을 한 줄씩 해석

class GradFn(ABC):

GradFn 클래스는 추상 기본 클래스(ABC)를 상속받는다. 이 클래스는 기울기 함수

를 위한 기반 클래스 역할을 한다.

```
def __init__(self, *args: 'TensorNode') -> None:  
  
    self.nodes: tuple['TensorNode', ...] = args
```

생성자는 가변 인자로 TensorNode 인스턴스들을 받아들이고, 이 인스턴스들을 self.nodes에 튜플로 저장한다.

```
def __call__(self, y: 'TensorNode') -> None:
```

```
    self.propagate(y)
```

`__call__` 메서드는 객체를 함수처럼 호출할 수 있게 해준다. 여기서는 `y` TensorNode를 인자로 받고, `self.propagate` 메서드를 호출한다.

`@abstractmethod`

```
def f_d(self, *args: 'TensorNode') -> tuple[ndarray, ...]:  
  
    ...
```

: `f_d`는 추상 메서드로, 각 파생 클래스에서 구현해야 한다. 이 메서드는 입력된 TensorNode들에 대한 함수의 편미분 값을 계산하고 반환한다.

`@staticmethod`

```
def _handle_broadcast(x: 'TensorNode', dx: ndarray) -> ndarray:
```

`handle_broadcast`라는 정적 메서드를 정의하고 있다. 이 메서드는 객체의 인스턴스나 클래스 변수에 접근하지 않기 때문에 `@staticmethod` 데코레이터를 사용한다. 이 메서드는 `x`라는 `TensorNode` 객체와 `dx`라는 `ndarray`를 인자로 받는다. `x`는 원래의 텐서 노드이며, `dx`는 이 텐서 노드에 대한 기울기이다. 이 메서드의 목적은 `dx`의 형태를 `x`의 형태에 맞추는 것이다.

```
if dx.ndim > x.ndim:
```

`dx`의 차원(ndim)이 `x`의 차원보다 큰지 확인한다. 이는 `dx`가 브로드캐스트된 결과로 인해 `x`보다 더 많은 차원을 가질 수 있음을 의미한다. 브로드캐스팅은 텐서 간 연산에서 차원이 맞지 않는 경우, 작은 텐서의 차원을 자동으로 맞추는 방식이다.

```
assert dx.shape[-x.ndim:] == x.shape or x.shape == ()
```

`dx`의 마지막 `x.ndim` 차원이 `x`의 차원과 동일한지 확인하는 `assert` 문이다. `x.shape` 가 ()일 때는 `x`가 스칼라임을 나타낸다. 이 조건이 성립하지 않으면, `dx`의 브로드 캐스트가 잘못되었다는 것을 의미하며, 이는 프로그램의 논리에 오류가 있음을 나타낸다.

```
dx = dx.reshape(-1, *x.shape).sum(0)
```

`dx`를 재구성하여 `x`의 차원에 맞춘다. `reshape(-1, *x.shape)`는 `dx`를 `x.shape`와 동일한 형태로 변환하고, 처음에 -1을 사용하여 첫 번째 차원의 크기를 자동으로 계산한다. `sum(0)`을 통해 첫 번째 차원에서 모든 값을 합쳐 `dx`의 차원을 `x`와 맞춘다.

```
else:
```

```
    assert dx.ndim == x.ndim
```

`dx.ndim`이 `x.ndim`보다 크지 않은 경우, 즉 두 배열의 차원이 같거나 `dx`의 차원이 더 작은 경우에 대한 처리이다. `dx`와 `x`의 차원이 같아야 한다는 것을 `assert`로 확인한다. 만약 이 조건이 만족되지 않으면, 차원이 다르다는 것은 불일치를 나타내며, 예상치 못한 오류가 발생했음을 나타낸다.

```
for i, (n_dx, n_x) in enumerate(zip(dx.shape, x.shape)):
```

`dx.shape`과 `x.shape`을 동시에 순회하면서 차원별로 대응하는 값을 가져온다. `enumerate`를 사용하여 각 차원의 인덱스 `i`와 `dx` 및 `x`의 해당 차원의 크기(`n_dx`와 `n_x`)를 가져온다.

```
if n_x == 1:
```

```
    dx = dx.sum(i, keepdims=True)
```

만약 `x`의 특정 차원(`n_x`)이 1이라면, `dx`의 해당 차원에 대해 합계를 구하고, 차원을 유지한다(`keepdims=True`). 이는 `dx`의 해당 차원에 대해 브로드캐스트된 차원

을 제거하여 x 와 동일한 차원으로 맞춘다. `keepdims=True`는 결과의 차원을 유지하여, dx 의 차원 수가 줄어들지 않게 한다.

```
return dx
```

형태가 조정된 dx 를 반환한다. 반환된 dx 는 x 와 같은 차원을 가지며, 이후의 계산에서 올바른 기울기를 전파할 수 있게 한다.

```
def propagate(self, y: 'TensorNode') -> None:
```

`propagate`라는 메서드를 정의하고 있다. 이 메서드는 y 라는 `TensorNode` 객체를 인자로 받는다. `self`는 `GradFn` 클래스의 인스턴스를 가리키며, y 는 기울기를 전파해야 할 대상 노드이다. \rightarrow `None`은 이 메서드가 아무것도 반환하지 않는다는 것을 명시적으로 나타낸다.

```
grads: tuple[ndarray, ...] = self.f_d(*self.nodes, y)
```

`self.f_d(*self.nodes, y)`를 호출하여 현재 노드들에 대한 기울기(편미분)를 계산한다. `*self.nodes`는 `self.nodes` 튜플의 각 요소를 개별 인자로 전달한다. 이 경우, `self.nodes`는 `TensorNode` 객체들을 포함하는 튜플이다. `f_d` 메서드는 연산의 편미분을 계산하는 메서드로, y 와 `self.nodes`를 사용하여 기울기를 계산한다. 계산된 기울기들은 `grads`라는 튜플에 저장되며, 이 튜플의 각 요소는 `ndarray` 타입이다. 각각의 `ndarray`는 연산 그래프에서 특정 입력에 대한 기울기를 의미한다.

```
for x, dx in zip(self.nodes, grads):
```

`zip(self.nodes, grads)`는 `self.nodes`와 `grads`를 한 쌍씩 묶어 튜플로 반환한다. `for` 루프를 통해 각 x (노드)와 해당 dx (기울기)를 순차적으로 처리한다. x 는 `self.nodes`에서 하나의 `TensorNode`를 가리키고, dx 는 해당 `TensorNode`에 대한 기울기 값을 의미한다.

```
if x.requires_grad:
```

`x.requires_grad`가 `True`인 경우에만 블록 내의 코드를 실행한다. 이는 해당 노드가 기울기 계산에 포함되어야 한다는 것을 나타낸다. `requires_grad`가 `True`라면, 역전파 과정에서 해당 노드의 기울기를 저장하고 업데이트할 필요가 있다.

```
if x.shape != dx.shape:
```

x.shape와 dx.shape가 다른지 확인한다. x는 TensorNode이며, x.shape는 그 내부의 배열(x.arr)의 형태를 나타낸다. dx는 x에 대한 기울기 배열로, dx.shape는 이 기울기 배열의 형태를 나타낸다. 만약 두 형태가 다르다면, 이는 브로드캐스팅(broadcasting) 때문에 발생할 수 있으며, 이에 따라 기울기 dx의 형태를 x.shape에 맞추기 위한 추가 조정이 필요하다.

```
dx = self._handle_broadcast(x, dx)
```

handle_broadcast 메서드를 호출하여 dx를 x.shape에 맞도록 조정한다. 이 메서드는 dx가 x의 형태에 맞도록 합산하거나, 필요한 경우 차원을 추가하여 맞춰준다. 이렇게 조정된 dx는 이후 기울기 업데이트 과정에서 올바르게 사용될 수 있다.

```
if x.grad is not None:
```

```
x.grad += dx
```

각 노드 x의 기울기 grad가 이미 존재하면 새로 계산된 기울기 dx를 더하고,

```
else:
```

```
x.grad = dx
```

그렇지 않으면 새로 할당한다.

```
x.grad_cnt -= 1
```

```
if x.grad_fn is not None and x.grad_cnt == 0:
```

```
x.grad_fn(x)
```

x.grad_cnt를 감소시킨 후, 모든 기울기가 계산되었고(grad_cnt가 0이 되었을 때), x의 기울기 함수(grad_fn)가 존재하면 해당 함수를 호출하여 추가적인 역전파를 수행한다. 이는 복잡한 계산 그래프에서 연쇄적인 역전파를 가능하게 한다.

2. f_d 가 staticmethod인 경우와 그렇지 않은 경우의 차이

Staticmethod 인 경우 인스턴스 상태나 클래스 상태에 접근하지 않고, 단지 입력된 파라미터만 사용하여 연산을 수행한다. `f_d` 에서는 인스턴스 변수에 접근할 필요가 없기 때문에 `staticmethod` 를 사용한다.

그렇지 않은 경우엔 메서드가 클래스의 인스턴스 변수에 접근하거나 수정할 필요가 있을 때 사용한다.

3. `AddGradFn` , `MulGradFn` , `DivGradFn` , `MatMulGradFn` 의 `f_d` 각각 해석하기 - 편미분에 대해 생각해보자.

AddGradFn에서, 두 입력의 편미분은 각각 y 의 기울기와 동일하다.

MulGradFn에서, 두 입력의 편미분은 각각 다른 입력에 대한 값과 y 의 기울기의 곱이다.

DivGradFn에서, 분모의 편미분은 분모의 제곱의 역수와 y 의 기울기의 곱에 음수를 취한 값이고, 분자는 분모의 역수와 y 의 기울기의 곱이다.

MatmulGradFn에서, 행렬 곱의 편미분은 각각 다른 입력의 전치와 y 의 기울기의 곱이다.

3. `RSubGradFn` , `RDivGradFn` , `RPowGradFn` , `RMatmulGradFn` 은 왜 `f_d` 가 따로 없을까? -상속에 대해 생각해보자

이 클래스들은 기본 연산의 역연산을 나타내기 때문에, 기본 연산의 `GradFn`을 상속받고, 생성자에서 입력의 순서만 바꿔서 기본 연산의 편미분 공식을 그대로 사용한다. 그래서 `f_d`가 따로 없는 것이다. 이러한 방식은 코드의 중복을 줄이고, 유지보수를 용이하게 한다.

4. `GetitemGradFn` , `SetitemGradFn` , `SetitemTensorGradFn` 해석하기

- `__getitem__` , `__setitem__` 도 미분 가능한 연산인가?

GetitemGradFn: 특정 키에 대해 인덱싱된 결과의 기울기는 해당 키 위치에 y 의 기울기를 설정하고, 나머지는 0이다.

SetitemGradFn: 특정 키에 값을 설정할 때, 원래 배열에서 해당 키를 제외한 위치의 기울기는 유지되고, 설정된 키 위치의 기울기는 0이 된다.

SetitemTensorGradFn: 특정 키에 다른 텐서의 값을 설정할 때, 설정값의 기울기와 원래 배열의 기울기를 적절히 처리한다. 이 메서드들은 `_getitem_`과 `_setitem_`이 미분 가능한 연산임을 보여준다. 이는 복잡한 텐서 연산에서도 세밀한 기울기 계산을 가능하게 하며, 딥러닝 라이브러리의 기능을 강화한다.

III. nn

1. Parameter, Module 의 구조

(1) Tensor와 Parameter의 차이

Tensor 는 다차원 배열을 처리하기 위한 기본 단위로, 데이터를 저장하고 수치 계산을 수행한다. 기울기를 필요로 하는 연산에서는 `requires_grad=True` 설정을 통해 자동 미분이 가능하다.

Parameter는 주로 신경망의 학습 가능한 가중치를 나타내며, Tensor를 상속받아 구현된다. Parameter 클래스는 항상 `requires_grad=True`로 설정되어 있어, 역전파 시 기울기를 계산하고 업데이트할 수 있다. 이는 신경망에서 학습이 필요한 파라미터를 정의할 때 사용된다.

(2) He initialization

He initialization은 ReLU 활성화 함수를 사용하는 신경망 계층의 가중치를 초기화하는 방법이다. 이 초기화 방법은 각 노드의 입력 연결 수의 역수의 제곱근에 기반하여 가중치를 초기화한다. 이는 입력의 분산을 1로 유지하고 역전파 시 gradient vanishing problem을 완화하는 데 도움을 준다.

(3) class Module의 모든 줄을 한 줄씩 해석

class Module:

Module 클래스를 정의하고 있다. 이는 신경망의 모든 구성 요소(레이어 등)를 표

현하기 위한 기본 클래스이다.

```
def _forward_unimplemented(*args, **kwargs) -> None:  
    raise Exception("forward not implemented")
```

: forward_unimplemented라는 내부 메서드를 정의하고 있다. 이 메서드는 forward 메서드가 구현되지 않았을 때 호출된다. *args와 **kwargs를 통해 임의의 위치 및 키워드 인수를 받을 수 있다. 메서드 내에서 예외를 발생시키며, "forward not implemented"라는 메시지를 포함한 Exception을 던진다. 이는 서브클래스에서 forward 메서드를 반드시 구현해야 한다는 것을 보장하기 위한 장치이다.

forward: Callable[..., Any] = _forward_unimplemented

: forward 메서드를 _forward_unimplemented로 초기화한다. 이는 forward 메서드가 서브클래스에서 구현되지 않으면 예외를 발생시키도록 한다. forward의 타입은 Callable[..., Any]로 정의되어 있어, 임의의 인수와 반환 타입을 가질 수 있음을 나타낸다.

def __call__(self, *args, **kwargs) -> Any:

```
    return self.forward(*args, **kwargs)
```

: __call__ 메서드를 정의한다. 이 메서드는 객체를 함수처럼 호출할 수 있게 한다. *args와 **kwargs를 통해 전달된 인수를 사용하여 forward 메서드를 호출하고 그 결과를 반환한다. 이는 모듈 객체가 직접 호출될 때, 자동으로 forward 메서드를 실행하도록 한다. 따라서, module_instance(input)처럼 사용하면, module_instance.forward(input)을 호출하는 것과 동일하게 동작한다.

def parameters(self) -> list[Parameter]:

parameters 메서드를 정의한다. 이 메서드는 모듈 내의 모든 학습 가능한 파라미터를 반환한다. 반환 타입은 list[Parameter]로, Parameter 객체를 포함하는 리스트이다.

```
params: list[Parameter] = []
```

params라는 빈 리스트를 생성한다. 이 리스트는 모듈 내의 모든 Parameter 객체를 저장하는 데 사용된다.

```
for v in self.__dict__.values():
```

self.__dict__.values()를 통해 현재 인스턴스의 모든 속성 값을 순회한다. __dict__는 객체의 속성을 저장하는 사전이다. 이 반복문을 통해 모듈 내의 모든 속성에 접근할 수 있다.

```
if isinstance(v, Module):
```

```
    params += v.parameters()
```

v가 Module 인스턴스인 경우, v.parameters()를 호출하여 하위 모듈의 파라미터 리스트를 가져온다. params += v.parameters()는 params 리스트에 하위 모듈의 파라미터를 추가한다. 이는 모듈 내에 포함된 모든 서브모듈의 파라미터를 재귀적으로 수집한다.

```
elif isinstance(v, Parameter):
```

```
    params.append(v)
```

v가 Parameter 인스턴스인 경우, params 리스트에 직접 추가한다. 이는 모듈 내의 직접적인 파라미터를 수집하는 부분이다.

```
return params
```

: 수집된 모든 파라미터를 포함하는 params 리스트를 반환한다.

(4) class Sequential 의 모든 줄을 한 줄씩 해석하세요!

```
class Sequential(Module):
```

```
    def __init__(self, *args) -> None:
```

```
        for i, module in enumerate(args):
```

```
            setattr(self, str(i), module)
```

: Sequential 클래스는 Module 클래스를 상속받는다. 생성자에서는 여러 모듈을 인자로 받고, 각 모듈을 인덱스를 키로 사용하여 현재 인스턴스에 저장한다.

```
def forward(self, x: Tensor) -> Tensor:  
    for layer in self.__dict__.values():  
        x = layer(x)  
  
    return x
```

: forward 메서드는 입력 x를 받아 각 계층을 순차적으로 통과시킨 결과를 반환한다.

2. ReLU , Sigmoid , Tanh , CrossEntropyLoss 가 굳이 Module 로 존재하는 이유

이러한 함수들이 Module 로 구현되어 있는 이유는 신경망 구조에서 이들 함수를 계층(layer)처럼 사용하기 위함이다. 모듈로 구현함으로써, 사용자는 이들 함수를 다른 신경망 계층과 동일하게 취급하고, 쉽게 구성할 수 있다. 예를 들어, 사용자는 Sequential 모듈 안에 ReLU 나 Sigmoid 를 다른 선형 계층(Linear)과 함께 추가하여, 신경망의 특정 부분에서 활성화 함수를 적용할 수 있다. 또한 이러한 방식은 모듈의 파라미터 관리 및 저장, 로딩 작업을 일관성 있게 처리할 수 있게 해준다.

IV. optim

class SGD 의 모든 줄을 한 줄씩 해석하세요!

```
class SGD:
```

: SGD 클래스를 정의한다. 이 클래스는 확률적 경사 하강법(SGD) 최적화 알고리즘을 구현한다.

```
def __init__(self, params: Sequence[Tensor], lr: float) -> None:
```

: 생성자 함수이다. params는 최적화를 적용할 텐서의 리스트를 받고, lr은 학습률(learning rate)을 나타낸다.

`self.params = params`

: 인스턴스 변수 params에 생성자에서 받은 텐서의 리스트를 할당한다.

`self.lr = lr`

: 인스턴스 변수 lr에 생성자에서 받은 학습률을 할당한다.

`def step(self) -> None:`

: step 메서드를 정의한다. 이 메서드는 각 파라미터에 대해 경사 하강법 한 단계를 수행한다.

`for param in self.params:`

: self.params 리스트에 있는 각 파라미터 param에 대해 반복한다.

`if param.grad is not None:`

: 현재 파라미터의 기울기(grad)가 None이 아닌 경우에만 다음 연산을 수행한다.

`param.arr -= param.grad * self.lr`

: 파라미터의 배열(arr)에서 기울기(grad)에 학습률(lr)을 곱한 값을 빼준다. 이는 파라미터를 갱신하는 연산이다.

`def zero_grad(self) -> None:`

: zero_grad 메서드를 정의한다. 이 메서드는 모든 파라미터의 기울기를 초기화한다.

`for param in self.params:`

self.params 리스트에 있는 각 파라미터 param에 대해 반복한다.

`param.grad = None`

: 각 파라미터의 기울기(grad)를 None으로 설정하여 초기화한다. 이는 다음 반복에서 기울기 누적을 방지하기 위함이다.

V. functions.py

1. sigmoid_naive 와 sigmoid 의 차이

sigmoid_naive 함수는 $\exp(-x)$ 를 계산하기 위해 자체적으로 정의된 \exp 함수를 사용한다. 이는 신경망에서 자주 사용되는 활성화 함수인 시그모이드 함수를 간단하게 구현한 것이다. 이 함수의 구현은 일반적으로 수치적으로 불안정할 수 있고, \exp 연산을 통해 오버플로우나 언더플로우가 발생할 위험이 있다. **Sigmoid** 함수는 `_new_tensor`를 통해 구현되어 있으며, `np.exp(-x.arr)`을 사용하여 구현되어 있다. NumPy의 `np.exp` 함수는 더욱 최적화되어 있으며, `_new_tensor` 호출을 통해 미분 가능한 텐서로 반환된다. 또한, `SigmoidGradFn`을 통해 역전파 시 사용될 그래디언트 함수가 정의되어 있어, 학습 과정에서 그래디언트를 자동으로 계산할 수 있다.

2. log 와 LogGradFn

log 함수는 입력 텐서 x 의 자연 로그를 계산한다. NumPy의 `np.log(x.arr)`를 사용하여 계산하고, `_new_tensor`를 통해 미분 가능한 텐서를 반환한다. **LogGradFn**은 `log` 함수의 미분을 처리하는 클래스이다. `log` 함수의 미분은 $1/x$ 이며, 이 그래디언트 함수는 역전파 과정에서 자동으로 적용되어 그래디언트를 계산한다.

3. sum 과 **SumGradFn**

Sum 함수는 입력 텐서 x 의 원소들을 선택적으로 지정된 축에 따라 합산한다. 이 때, `keepdim` 옵션을 통해 결과 텐서의 차원을 유지할 수 있다. **SumGradFn**은 `sum` 함수의 미분을 처리하는 클래스이다. `sum` 연산의 미분은 각 입력 원소에 대해 동일하게 1을 반환하는 형태이다.

4. relu 와 **ReLUGradFn**

Relu 함수는 입력 텐서의 각 원소에 대해 0과의 최대값을 취하는 Rectified Linear Unit 활성화 함수이다. 널리 사용되는 활성화 함수 중 하나이다. **ReLUGradFn**은 `relu` 함수의 미분을 처리한다. `relu`의 미분은 입력이 0보다 클 때 1, 그렇지 않으면 0이다.

5. repeat 와 **RepeatGradFn**

Repeat 함수는 입력 텐서 x 의 원소들을 지정된 횟수만큼 반복하여 크기를 늘린다. `np.tile`을 사용하여 구현된다. **RepeatGradFn**은 `repeat` 함수의 미분을 처리하는 클래스이다. `repeat` 연산의 미분은 반복된 각 원소에 대해 그래디언트를 적절히 분배하는 과정을 포함한다.

6. main.py (numpytorch 밖에 있는 놈)

1. F1 Score에 대한 간단한 설명

F1 점수는 정밀도(precision)와 재현율(recall)의 조화 평균으로 계산된다. 이 두 측정값은 각각 모델이 얼마나 정확하게 긍정 사례를 예측하는지, 실제 긍정 사례를 얼마나 잘 찾아내는지를 나타낸다. F1 점수는 이 두 측정치를 동시에 고려하여, 데이터 라벨의 균형을 잡을 때 특히 유용하다. 점수는 0(최악)에서 1(최선) 사이의 값으로 표현된다.

2. MNIST Dataset에 대한 간단한 설명

MNIST 데이터셋은 손으로 쓴 숫자(0부터 9까지)의 이미지를 포함하는 데이터셋으로, 컴퓨터 비전 분야에서 널리 사용되는 기준 테스트 데이터셋이다. 이 데이터셋에는 28x28 픽셀 크기의 흑백 이미지 70,000개가 포함되어 있으며, 이 중 60,000개는 훈련용이고 10,000개는 테스트용으로 사용된다.

3. model, criterion, optimizer의 선언

```
model = MNISTClassificationModel()
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.SGD(model.parameters(), lr)
```

model은 MNISTClassificationModel의 인스턴스로, MNIST 숫자를 분류하기 위한 신경망 모델이다.

criterion은 손실 함수로 CrossEntropyLoss를 사용하여 모델의 예측과 실제 레이블 간의 차이를 측정한다.

optimizer는 SGD(확률적 경사 하강법) 최적화기로, 모델 파라미터를 업데이트하는 데 사용된다. lr은 학습률을 지정한다.

4. 학습 루프 설명

(1) optimizer.zero_grad()의 역할과 필요한 이유

optimizer.zero_grad()는 각 훈련 스텝의 시작에서 호출되어 이전 스텝에서 계산된 기울기를 초기화한다. 이는 기울기가 누적되는 것을 방지하여 각 배치의 독립적인 학습을 보장한다.

(2) logits = model(x)

model(x)는 입력 데이터 x를 모델에 통과시켜 로짓값을 계산한다. 이 값은 신경망의 마지막 선형 레이어에서의 출력값으로, 활성화 함수에 의해 변환되기 전의 원시 예측 값이다.

(3) **loss = criterion(logits, y)**

criterion(logits, y)는 예측된 로짓과 실제 레이블 y 사이의 손실을 계산한다. criterion이 callable한 이유는 nn.CrossEntropyLoss 클래스가 함수처럼 동작하도록 설계되었기 때문이다. loss의 shape는 보통 스칼라 값이다.

(4) **loss.backward()**

loss.backward()는 손실 함수에서 계산된 결과를 바탕으로 모델 파라미터에 대한 기울기를 역전파한다.

(5) **optimizer.step()**

optimizer.step()은 계산된 기울기를 사용하여 모델 파라미터를 업데이트한다. 이는 학습률과 기울기에 따라 파라미터를 조정함으로써 학습 과정을 진행한다.

(6) **macro, micro = val(model, x_val, y_val)**

val 함수는 검증 데이터셋에 대해 모델을 평가하고, 매크로와 마이크로 F1 점수를 계산하여 반환한다. 이는 모델의 성능을 측정하는 지표로 사용된다.

macro F1 스코어는 각 클래스의 F1 점수를 계산한 후, 이 점수들의 산술 평균을 구하는 방식이다. 모든 클래스에 동일한 가중치를 부여한다. 즉, 각 클래스가 데이터셋에서 차지하는 비율에 관계없이 동일하게 취급된다. 이는 클래스 불균형이 큰 데이터셋에서 각 클래스의 성능을 개별적으로 평가하고자 할 때 유용하다. 클래스의 크기가 크게 차이나는 경우에도 모든 클래스의 중요도를 동일하게 본다.

micro F1 스코어는 전체 데이터셋을 기반으로 TP(True Positive), FP(False Positive), FN(False Negative)의 총합을 사용하여 정밀도와 재현율을 계산하고, 이로부터 F1 점수를 계산하는 방식이다. 각 샘플에 동일한 가중치를 부여한다. 즉, 모든 샘플이 동일하게 고려되며, 클래스의 크기에 따라 더 많은 영향을 받는다. 전체적인 시스템의 성능을 평가할 때 유용하다. 클래스 간의 데이터 불균형이 있을 때, 모델이 전체적으로 얼마나 잘 동작하는지를 파악하는 데 효과적이다.

정리하면, macro는 모든 클래스에 대해 동등한 관심을 가질 때 유용하며, 클래스 불균형이 있는 데이터셋에서 개별 클래스의 성능을 평가하는 데 적합하고, micro는 전체 데이터셋의 성능을 중시할 때 유용하며, 특히 클래스 간 불균형이 있는 데이터셋에서 전체적인 모델 성능을 평가하는 데 적합하다.

(7) 이중 모델 파라미터가 실제로 업데이트되는 건 언제일까요?

모델 파라미터는 `optimizer.step()` 메서드가 호출될 때 실제로 업데이트된다. 이 과정에서 기울기 정보와 학습률을 활용하여 파라미터를 조정하며, 이는 모델 학습의 핵심 과정이다. 이 업데이트는 각 미니 배치(batch)마다 발생하며, 이를 통해 모델이 점진적으로 학습해 나간다.

2번 과제: CNN 구현하기, 구현한 CNN 코드에 대한 상세한 설명

```
# assignment.py

import numpytorch as npt
from numpytorch import Tensor, nn
from numpytorch import reshape
import numpy as np

class Conv2d(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int) -> None:
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        # Initialize weights and biases with numpy
        self.weights = npt.tensor(np.random.rand(out_channels, in_channels,
        kernel_size, kernel_size) * np.sqrt(2. / (in_channels * kernel_size *
        kernel_size)), requires_grad=True)
        self.bias = npt.tensor(np.zeros(out_channels), requires_grad=True)

    def forward(self, x: Tensor) -> Tensor:
        # x: (batch_size, in_channels, height, width)
        batch_size, _, height, width = x.shape
        kh, kw = self.kernel_size, self.kernel_size
        oh, ow = height - kh + 1, width - kw + 1

        # Output feature map
        out = np.zeros((batch_size, self.out_channels, oh, ow))

        for i in range(oh):
            for j in range(ow):
                # Get the current region
                x_region = x[:, :, i:i+kh, j:j+kw]
                # Perform convolution
                # Ensure tensors are compatible
                out[:, :, i, j] = np.tensordot(x_region.arr, self.weights.arr,
                axes=([1, 2, 3], [1, 2, 3])) + self.bias.arr

        return npt.tensor(out)

class MaxPool2d(nn.Module):
    def __init__(self, kernel_size: int, stride: int) -> None:
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride
```

```

def forward(self, x: Tensor) -> Tensor:
    # x: (batch_size, in_channels, height, width)
    batch_size, channels, height, width = x.shape
    kh, kw = self.kernel_size, self.kernel_size
    sh, sw = self.stride, self.stride

    oh, ow = (height - kh) // sh + 1, (width - kw) // sw + 1
    out = np.zeros((batch_size, channels, oh, ow))

    for i in range(oh):
        for j in range(ow):
            x_region = x[:, :, i*sh:i*sh+kh, j*sw:j*sw+kw]
            out[:, :, i, j] = np.max(x_region, axis=(2, 3))

    return npt.tensor(out)

class MNISTClassificationModel(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = Conv2d(1, 32, kernel_size=3) # 28x28 -> 26x26
        self.pool1 = MaxPool2d(kernel_size=2, stride=2) # 26x26 -> 13x13
        self.conv2 = Conv2d(32, 64, kernel_size=3) # 13x13 -> 11x11
        self.pool2 = MaxPool2d(kernel_size=2, stride=2) # 11x11 -> 5x5
        self.fc1 = nn.Linear(64 * 5 * 5, 128)
        self.fc2 = nn.Linear(128, 10, bias=False)

    def forward(self, x: Tensor) -> Tensor:
        x = self.conv1(x)
        x = npt.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = npt.relu(x)
        x = self.pool2(x)
        x = reshape(x, (x.shape[0], -1)) # Flatten
        x = self.fc1(x)
        x = npt.relu(x)
        logits = self.fc2(x)
        return logits

```

다음 장에 실행 결과 캡처샷과 설명이 있다.

`n_iter=5000`으로 설정했으나, 2000에서 이미 0.9를 넘은 모습이다.

```
File Edit Selection View Go ... 1workspace1 Explorer 1WORKSPACE1 8-1-MLP main.py assignment.py 37 38 39 if __name__ == '__main__': 40 x_train, y_train, x_val, y_val = get_mnist() 41 42 ### edits allowed here ### 43 n_batch = 32 44 n_iter = 5000 45 n_print = 100 46 n_val = 2000 47 lr = 1e-03 ##### 48 49 model = MNISTClassificationModel() 50 criterion = nn.CrossEntropyLoss() 51 optimizer = optim.SGD(model.parameters(), lr) 52 53 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE Python - 8-1-MLP + ... 1699/5000 [03:21<06:04, 9.05it/s] 1799/5000 [03:33<05:54, 9.03it/s] 1899/5000 [03:45<05:32, 9.31it/s] 1999/5000 [03:56<05:33, 9.01it/s] 2099/5000 [04:40<05:43, 8.45it/s] 2199/5000 [04:51<05:00, 9.31it/s] 2299/5000 [05:02<04:58, 9.28it/s] 2399/5000 [05:14<05:04, 8.53it/s] OUTLINE TIMELINE 339301815017607 In 46, Col 17 Spaces: 4 UTF-8 LF Python 3.11.9 (base:conda) Prettier
```

조금 더 돌려봤지만, 4000번 돌았음에도 거의 점수가 오르지 않았다.

```
File Edit Selection View Go ... 1workspace1 Explorer 1WORKSPACE1 8-1-MLP main.py assignment.py 37 38 39 if __name__ == '__main__': 40 x_train, y_train, x_val, y_val = get_mnist() 41 42 ### edits allowed here ### 43 n_batch = 32 44 n_iter = 5000 45 n_print = 100 46 n_val = 2000 47 lr = 1e-03 ##### 48 49 model = MNISTClassificationModel() 50 criterion = nn.CrossEntropyLoss() 51 optimizer = optim.SGD(model.parameters(), lr) 52 53 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE Python - 8-1-MLP + ... 3599/5000 [07:46<02:49, 8.26it/s] 3699/5000 [07:47<02:35, 8.39it/s] 3799/5000 [07:59<02:16, 8.83it/s] 3899/5000 [08:10<02:13, 8.23it/s] 3999/5000 [08:22<02:05, 7.96it/s] 4099/5000 [09:08<01:45, 8.58it/s] 4199/5000 [09:19<01:30, 8.84it/s] 4299/5000 [09:31<01:21, 8.60it/s] 4399/5000 [09:42<01:06, 9.07it/s] OUTLINE TIMELINE 303580843666812776 In 46, Col 17 Spaces: 4 UTF-8 LF Python 3.11.9 (base:conda) Prettier
```

5000번까지 iteration을 다 돌고 난 후, macro micro 모두 0.9를 넘은 모습이다.

The screenshot shows a Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows a tree view of the workspace. The "1 WORKSPACE1" section contains files like `ipython_checkpoints`, `.mypy_cache`, `vscode`, and several notebooks and scripts. The "8-1-MLP" folder is expanded, showing files `main.py` and `assignment.py`.
- Code Editor:** Displays the content of `main.py`. The code imports `macro` and `micro` from `assignment.py`. It defines a function `get_mnist()` and sets parameters for a neural network model. The `model` is initialized as `MNISTClassificationModel()` with a cross-entropy loss criterion and an SGD optimizer.
- Terminal:** Shows the output of a Python script. The output includes numerical values and progress indicators, such as "final score", "macro: 0.91764043 micro: 0.91750000", and "PS C:\Users\kg051\workspace1\workspace1\8-1-MLP>".
- Bottom Status Bar:** Shows the current file is `main.py`, the Python Debugger is active, and the connection status.

Mypy test까지 문제없이 통과한 모습이다.



```
File Edit Selection View Go ... ← → 1workspace1  
EXPLORER 1-WORKSPACE1 8-1-MLP assignment.py x  
ipynb_checkpoints mypy_cache vscode 1-신규기수프로젝트 2-Python(1) 3-2-Python(2) 4-1-Linux 5-1-Network 5-2-Web 5-3-Docker 6-1-Array-programming 6-3-ML 7-1-pandas 7-2-DB 7-3-Data-Visualization 8-1-MLP _pycache_ ipnb_checkpoints mypy_cache data numpytorch assignment.py main.py pyproject.toml requirements.txt 8-2-DL 9-2-NLP pytorch 배신 흥계학전 baseline_inimb OUTLINE TIMELINE  
assignment.py x  
85  
86 class MNISTClassificationModel(nn.Module):  
87     def __init__(self) -> None:  
88         super().__init__()  
89         self.conv1 = Conv2d(1, 32, kernel_size=3) # 28x28 -> 26x26  
90         self.pool1 = MaxPool2d(kernel_size=2, stride=2) # 26x26 -> 13x13  
91         self.conv2 = Conv2d(32, 64, kernel_size=3) # 13x13 -> 11x11  
92         self.pool2 = MaxPool2d(kernel_size=2, stride=2) # 11x11 -> 5x5  
93         self.fc1 = nn.Linear(64 * 5 * 5, 128)  
94         self.fc2 = nn.Linear(128, 10, bias=False)  
95  
96     def forward(self, x: Tensor) -> Tensor:  
97         x = self.conv1(x)  
98         x = nn.ReLU(x)  
99         x = self.pool1(x)  
100        x = self.conv2(x)  
101        x = nn.ReLU(x)  
102        x = self.pool2(x)  
103  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL CONSOLE  
Python - 8-1-MLP + ⚡ 🌐 ⚙️  
return func(self, options, args)  
~~~~~  
File "c:\users\gu051\anaconda3\lib\site-packages\pip\_internal\commands\install.py", line 483, in run  
    installed_versions[distribution.canonical_name] = distribution.version  
~~~~~  
File "c:\users\gu051\anaconda3\lib\site-packages\pip\_internal\metadata\importlib_dists.py", line 168, in version  
    return parse_version(self.dist.version)  
~~~~~  
File "c:\users\gu051\anaconda3\lib\site-packages\pip\_internal\vendor\packaging\version.py", line 56, in parse  
    return Version(version)  
~~~~~  
File "c:\users\gu051\anaconda3\lib\site-packages\pip\_internal\vendor\packaging\version.py", line 200, in __init__  
    match = self._regex.search(version)  
~~~~~  
TypeError: expected string or bytes-like object, got 'NoneType'  
PS C:\Users\gu051\workspace\8-1-MLP> mypy .  
Success: no issues found in 18 source files  
PS C:\Users\gu051\workspace\8-1-MLP>
```

```
# assignment.py
```

```
import numpytorch as npt
from numpytorch import Tensor, nn
from numpytorch import reshape
import numpy as np
```

numpytorch라는 커스텀 라이브러리와 그 내부의 Tensor, nn 모듈, reshape 함수를 가져온다. numpytorch는 NumPy를 기반으로 한 딥러닝 프레임워크로 보인다. NumPy는 수치 계산을 위한 파이썬 라이브러리로, 배열 연산을 효율적으로 처리 할 수 있도록 한다.

```
class Conv2d(nn.Module):
```

Conv2d 클래스는 nn.Module을 상속받아 2D 합성곱 연산을 구현하는 클래스이다.

```
def __init__(self, in_channels: int, out_channels: int, kernel_size: int) -> None:
    super().__init__()
```

`__init__` 메서드는 클래스 생성자이다. 입력 채널 수, 출력 채널 수, 커널 크기를 매개변수로 받는다. `super().__init__()`는 부모 클래스인 `nn.Module`의 초기화 메서드를 호출한다.

```
self.in_channels = in_channels
```

```
self.out_channels = out_channels
```

```
self.kernel_size = kernel_size
```

입력 채널 수, 출력 채널 수, 커널 크기를 인스턴스 변수로 저장한다.

```

# Initialize weights and biases with numpy

self.weights = np.tensor(np.random.rand(out_channels, in_channels,
kernel_size, kernel_size) * np.sqrt(2. / (in_channels * kernel_size * kernel_size)),
requires_grad=True)

self.bias = np.tensor(np.zeros(out_channels), requires_grad=True)

```

weights는 out_channels, in_channels, kernel_size, kernel_size 차원의 랜덤 초기화된 가중치 텐서이다. 이 가중치는 He 초기화를 사용하여 초기화된다. requires_grad=True로 설정하여 학습 가능하도록 한다. bias는 out_channels 길이의 0으로 초기화된 바이어스 텐서이다. 역시 requires_grad=True로 설정되어 있다.

```

def forward(self, x: Tensor) -> Tensor:

forward 메서드는 입력 텐서 x를 받아 출력 텐서를 반환하는 함수이다. 이 메서드는 합성곱 연산의 핵심 부분이다.

# x: (batch_size, in_channels, height, width)

batch_size, _, height, width = x.shape

kh, kw = self.kernel_size, self.kernel_size

oh, ow = height - kh + 1, width - kw + 1

```

입력 텐서 x의 배치 크기, 채널 수, 높이, 너비를 추출한다. 커널의 높이 kh와 너비 kw는 동일하게 커널 크기로 설정한다. 출력 특성 맵의 높이 oh와 너비 ow를 계산한다. 이는 입력 크기에서 커널 크기를 빼고 1을 더한 값이다.

```

# Output feature map

out = np.zeros((batch_size, self.out_channels, oh, ow))

```

합성곱 연산 결과를 저장할 출력 배열 out을 0으로 초기화한다. 출력 배열의 차원은 (batch_size, out_channels, oh, ow)이다.

```

for i in range(oh):
    for j in range(ow):
        # Get the current region
        x_region = x[:, :, i:i+kh, j:j+kw]
        # Perform convolution
        # Ensure tensors are compatible
        out[:, :, i, j] = np.tensordot(x_region.arr, self.weights.arr, axes=[[1, 2, 3], [1, 2, 3]]) + self.bias.arr

```

출력 높이와 너비에 대해 이중 루프를 돌며 합성곱 연산을 수행한다. x_region은 현재 커널 위치에서 입력 텐서의 부분 영역이다. np.tensordot를 사용하여 x_region과 weights 간의 텐서 내적을 계산하고, bias를 더하여 out의 현재 위치에 저장한다.

```
return npt.tensor(out)
```

계산된 출력 배열을 numpytorch의 Tensor로 변환하여 반환한다.

```
class MaxPool2d(nn.Module):
```

MaxPool2d 클래스는 2D 최대 풀링 연산을 구현하는 클래스이다

```

def __init__(self, kernel_size: int, stride: int) -> None:
    super().__init__()

```

def __init__(...): 클래스의 생성자 메서드를 정의한다. 최대 풀링의 커널 크기와 스트라이드를 인자로 받는다.

`super().__init__()`: 부모 클래스인 nn.Module의 초기화 메서드를 호출한다.

```
self.kernel_size = kernel_size
```

```
self.stride = stride
```

최대 풀링의 커널 크기를 `self.kernel_size`라는 인스턴스 변수에 저장한다. 최대 풀링의 스트라이드를 `self.stride`라는 인스턴스 변수에 저장한다.

```
def forward(self, x: Tensor) -> Tensor:
```

`forward` 메서드를 정의한다. 이는 `Tensor` 타입의 입력 `x`를 받아 최대 풀링을 수행한 후 `Tensor` 타입의 출력을 반환한다.

```
# x: (batch_size, in_channels, height, width)
```

```
batch_size, channels, height, width = x.shape
```

입력 텐서의 배치 크기, 채널 수, 높이, 너비를 추출한다.

```
kh, kw = self.kernel_size, self.kernel_size
```

풀링 커널의 높이와 너비를 설정한다.

```
sh, sw = self.stride, self.stride
```

풀링 스트라이드의 높이와 너비를 설정한다.

```
oh, ow = (height - kh) // sh + 1, (width - kw) // sw + 1
```

```
out = np.zeros((batch_size, channels, oh, ow))
```

출력 특성 맵의 높이와 너비를 계산하고, 이를 기반으로 0으로 초기화된 출력 배열 out을 생성한다.

```
for i in range(oh):
```

```
    for j in range(ow):
```

출력 높이와 너비에 대해 이중 루프를 돌며 최대 풀링 연산을 수행한다.

```
x_region = x[:, :, i*sh:i*sh+kh, j*sw:j*sw+kw]
```

```
out[:, :, i, j] = np.max(x_region.arr, axis=(2, 3))
```

x_region은 현재 풀링 위치에서 입력 텐서의 부분 영역이다.

np.max를 사용하여 x_region에서 2차원 영역(높이와 너비) 내에서 최대 값을 계산하여 out의 현재 위치에 저장한다.

```
return npt.tensor(out)
```

계산된 출력 배열을 numpytorch의 Tensor로 변환하여 반환한다.

```
class MNISTClassificationModel(nn.Module):
```

MNISTClassificationModel 클래스는 전체 CNN 모델을 정의하는 클래스이다. 이는 nn.Module을 상속받는다.

```
def __init__(self) -> None:
```

```
    super().__init__()
```

__init__ 메서드는 모델의 구성 요소를 초기화한다. super().__init__()는 부모 클래스의 초기화 메서드를 호출한다.

```
self.conv1 = Conv2d(1, 32, kernel_size=3) # 28x28 -> 26x26
```

conv1은 입력 채널 1, 출력 채널 32, 커널 크기 3x3의 합성곱 계층이다. 입력 이미지 크기를 28x28에서 26x26으로 줄인다.

```
self.pool1 = MaxPool2d(kernel_size=2, stride=2) # 26x26 -> 13x13
```

pool1은 커널 크기 2x2, 스트라이드 2의 최대 풀링 계층이다. 이미지 크기를 26x26에서 13x13으로 줄인다.

```
self.conv2 = Conv2d(32, 64, kernel_size=3) # 13x13 -> 11x11
```

conv2는 입력 채널 32, 출력 채널 64, 커널 크기 3x3의 합성곱 계층이다. 이미지 크기를 13x13에서 11x11로 줄인다.

```
self.pool2 = MaxPool2d(kernel_size=2, stride=2) # 11x11 -> 5x5
```

pool2는 커널 크기 2x2, 스트라이드 2의 최대 풀링 계층이다. 이미지 크기를 11x11에서 5x5로 줄인다.

```
self.fc1 = nn.Linear(64 * 5 * 5, 128)
```

fc1은 64개의 5x5 특징맵을 입력으로 받고, 128개의 출력을 생성하는 완전 연결 계층이다.

```
self.fc2 = nn.Linear(128, 10, bias=False)
```

fc2는 128개의 입력을 받고, 10개의 출력을 생성하는 완전 연결 계층이다. 이는 최종 분류를 위한 출력 계층이다.

```
def forward(self, x: Tensor) -> Tensor:
```

forward 메서드는 입력 텐서 x 를 모델에 통과시켜 최종 출력을 반환한다.

```
x = self.conv1(x)
```

```
x = np.tanh(x)
```

입력 x 를 conv1 계층에 통과시킨 후, tanh 활성화 함수를 적용한다.

```
x = self.pool1(x)
```

x 를 pool1 계층에 통과시켜 최대 풀링을 적용한다.

```
x = self.conv2(x)
```

```
x = np.tanh(x)
```

x 를 conv2 계층에 통과시킨 후, tanh 활성화 함수를 적용한다.

```
x = self.pool2(x)
```

x 를 pool2 계층에 통과시켜 최대 풀링을 적용한다.

```
x = reshape(x, (x.shape[0], -1)) # Flatten
```

x 의 형태를 (batch_size, -1)로 변환하여 평탄화(flatten)한다. 이는 모든 특징을 1차원 벡터로 변환한다.

```
x = self.fc1(x)
```

```
x = np.t.relu(x)
```

평탄화된 x 를 $fc1$ 계층에 통과시킨 후, ReLU 활성화 함수를 적용한다.

```
logits = self.fc2(x)
```

x 를 $fc2$ 계층에 통과시켜 최종 출력 $logits$ 을 계산한다. $logits$ 은 각 클래스에 대한 점수를 나타낸다.

```
return logits
```

최종 계산된 $logits$ 을 반환한다. 이는 모델의 예측 결과로 사용된다.