report

1. Tensor 및 관련 개념 설명

1.1 TensorNode의 각 속성 설명

1.1.1 `TensorNode.arr`

- `arr`는 `TensorNode`가 실제로 저장하고 있는 데이터를 의미하며, 일반적으로 `numpy` 배열로 구현됨. 이 속성은 텐서의 실제 값을 담고 있으며, 모든 텐서 연산의 기본이 됨.

1.1.2 `TensorNode.requires_grad`

- `requires_grad`는 해당 텐서가 역전파 과정에서 그래디언트(gradient)를 계산해야 하는지 여부를 나타냄.

1.1.3 `TensorNode.is_leaf`

- `is_leaf`는 해당 텐서가 연산 그래프의 리프 노드인지 여부를 나타냄. 리프 노드는 부모 노드가 없는, 독립적인 텐서를 의미함.

1.1.4 `is_leaf`와 `requires_grad`의 관계

- `is_leaf`와 `requires_grad`는 서로 독립적인 속성임. 가능한 조합:
  1. `is_leaf=True`, `requires_grad=True`
  2. `is_leaf=True`, `requires_grad=False`
  3. `is_leaf=False`, `requires_grad=True`
  4. `is_leaf=False`, `requires_grad=False`

1.1.5 `TensorNode.grad_fn`

- `grad_fn`은 이 텐서가 생성된 연산을 참조함. 이 속성은 역전파 과정에서 각 연산에 대한 그래디언트를 계산할 때 사용됨.

1.1.6 `TensorNode.grad`

- `grad`는 역전파 과정에서 계산된 그래디언트를 저장하는 속성임. `requires_grad=True`로 설정된 경우에만 사용됨.

1.1.7 `TensorNode.grad_cnt`

- `grad_cnt`는 해당 텐서에 대해 그래디언트가 누적된 횟수를 나타내는 속성임.

1.2 TensorNode의 주요 메서드 해석

1.2.1 `TensorNode.backward`

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

- 이 메서드는 `TensorNode` 객체의 역전파를 시작하는 역할을 함. 반환값은 없음(`None`).

```
    assert self.grad_fn is not None
```

- 역전파를 수행하기 전에, `grad_fn`이 `None`이 아닌지 확인함. `grad_fn`이 없으면 역전파가 불가능함.

```
    assert self.shape == ()
```

- 이 줄은 텐서의 `shape`가 스칼라(`()`, 즉 단일 값)인지 확인함. 역전파는 보통 스칼라 값(예: 손실 함수의 출력)에서 시작되기 때문에, 이 조건이 필요함.

```
    self.grad = np.ones(())
```

- `grad`에 스칼라 값 1을 할당함. 이는 역전파가 시작될 때 그래디언트가 1로 초기화된다는 것을 의미함.

```
    self.grad_fn(self)
```

- `grad_fn`을 호출하여 현재 텐서의 그래디언트를 계산함. `grad_fn`은 역전파 중 각 연산에 대해 그래디언트를 계산하는 함수임.

1.2.2 `TensorNode._create_new_tensornode`

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

- 이 메서드는 새로운 `TensorNode`를 생성함. 입력으로는 `o`(다른 텐서 또는 값), `operation`(Numpy 연산 함수), `grad_fn`(그래디언트를 계산하는 함수 클래스)를 받음.

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

- 입력 `o`가 `TensorNode`가 아니면, `TensorNode` 객체로 변환함. 이 과정에서 일반 숫자나 Numpy 배열도 `TensorNode`로 변환됨.

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

- `operation` 함수를 사용해 `self`와 `o`의 `arr`(데이터) 간의 연산을 수행하고, 결과를 `new_arr`에 저장함.

```
    if self.requires_grad or o.requires_grad:
        new_requires_grad = True
        new_is_leaf = False
        new_grad_fn = grad_fn(self, o)
```

- `self` 또는 `o` 중 하나라도 `requires_grad=True`인 경우, 새로 생성될 텐서 노드의 `requires_grad`를 `True`로 설정하고, 이 노드는 리프 노드가 아니므로 `new_is_leaf`를 `False`로 설정함. 또한, `grad_fn`을 초기화함.

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

- `self`와 `o` 중 `requires_grad=True`인 경우 각각의 `grad_cnt`를 1씩 증가시킴. 이는 해당 노드가 그래디언트를 필요로 하는 연산에 포함되었음을 나타냄.

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

- 만약 `self`와 `o`가 모두 `requires_grad=False`인 경우, 새로운 텐서 노드는 그래디언트를 계산하지 않으며, 리프 노드로 간주됨. 따라서 `grad_fn`도 필요하지 않으므로 `None`으로 설정됨.

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

- 앞서 설정한 `new_arr`, `new_requires_grad`, `new_is_leaf`, `new_grad_fn`을 사용해 새로운 `TensorNode` 객체를 생성함.

```
    return new_TensorNode
```

- 새로 생성된 `TensorNode` 객체를 반환함.

1.2.3 `TensorNode._operation`

- 이 메서드는 텐서 간의 연산을 수행하는 기능을 담당함.

1.2.4 `TensorNode._assert_not_leaf`

- 이 메서드는 현재 텐서가 리프 노드가 아닌지를 확인하는 역할을 함.

1.3 Tensor.setitem 해석

`Tensor` 클래스의 `__setitem__` 메서드는 텐서 객체에서 특정 위치에 값을 설정하는 역할을 함. 이 메서드는 `TensorNode`의 `__setitem__`과는 다른 방식으로 구현되어 있으며, `Tensor`가 연산 그래프의 일부로 사용될 때 중요한 역할을 함.

```
def __setitem__(self, key, value: Value) -> None:
```

- `__setitem__` 메서드는 `Tensor` 객체의 특정 위치에 값을 설정함. `key`는 인덱스나 슬라이스, `value`는 설정하려는 값임. `value`는 `Tensor`이거나 일반 값일 수 있음.

```
    if self.requires_grad:
```

- 먼저, `self.requires_grad`가 `True`인지 확인함. 이 속성이 `True`라면, 현재 텐서가 그래디언트를 필요로 하고, 따라서 리프 노드일 수 없다는 의미임.

```
        assert not self.is_leaf
```

- `self.is_leaf`가 `False`이어야 함을 보장하기 위해 `assert` 문을 사용함. 만약 이 텐서가 리프 노드라면, 값을 직접 설정하는 것이 허용되지 않음. 리프 노드는 주로 학습 가능한 파라미터를 나타내며, 직접 수정해서는 안 됨.

```
        new_node = TensorNode(self.node, requires_grad=True,
is_leaf=False)
```

- 새로운 `TensorNode`를 생성함. 이 새로운 노드는 기존 노드의 정보를 기반으로 하지만, `is_leaf`가 `False`로 설정되어, 더 이상 리프 노드가 아님.

```
        if isinstance(value, Tensor):
            new_node.arr[key] = value.arr
```

- `value`가 `Tensor` 객체인 경우, 해당 `Tensor`의 `arr` 값을 `new_node`의 `arr` 속성의 지정된 `key` 위치에 할당함.

```
            if value.requires_grad:
                new_node.grad_fn = SetitemTensorGradFn(self.node,
value.node, key)
                value.node.grad_cnt += 1
```

- `value`가 `requires_grad=True`라면, `SetitemTensorGradFn`을 사용해 `grad_fn`을 설정함. 이는 현재 텐서에 대해 그래디언트를 계산할 수 있도록 설정하는 것임. 또한, `value.node.grad_cnt`를 1 증가시켜 이 노드가 그래디언트를 필요로 하는 연산에 포함되었음을 표시함.

```
            else:
                new_node.grad_fn = SetitemGradFn(self.node, key)
```

- `value`가 `requires_grad=False`인 경우, 간단하게 `SetitemGradFn`을 사용하여 `grad_fn`을 설정함.

```
        else:
            new_node.arr[key] = value
            new_node.grad_fn = SetitemGradFn(self.node, key)
```

- `value`가 `Tensor`가 아니라 일반 값인 경우, 해당 값을 `new_node.arr[key]`에 할당하고, `SetitemGradFn`을 `grad_fn`으로 설정함.

```
        self.node.grad_cnt += 1
        self.node = new_node
```

- 현재 노드의 `grad_cnt`를 1 증가시킴. 이 노드는 이제 새로운 연산의 결과로 생성되었음을 나타냄. 이후, `self.node`를 새로 생성된 `new_node`로 업데이트함.

```
    else:
        if isinstance(value, Tensor):
            self.node.arr[key] = value.node.arr
```

```
        else:
            self.node.arr[key] = value
```

- 만약 `self.requires_grad=False`라면, 그래디언트를 계산할 필요가 없으므로, 단순히 `self.node.arr[key]`에 값을 할당함. `value`가 `Tensor`이면 그 `arr` 값을 할당하고, 그렇지 않으면 `value`를 그대로 할당함.

왜 `TensorNode`가 아닌 `Tensor`에 구현되어 있는가?

- `__setitem__` 메서드는 `Tensor` 클래스에 구현되어 있음. 이는 `Tensor`가 연산의 추상화를 담당하기 때문임. `TensorNode`는 실제 데이터를 담고 있고, 그래디언트 계산을 위한 메타데이터를 관리함. 반면, `Tensor` 클래스는 이러한 노드들을 래핑하여 사용자에게 보다 직관적인 인터페이스를 제공함.
- `Tensor`는 연산 그래프의 각 노드를 관리하면서, 최종적으로 계산된 결과를 나타내는 객체임. `__setitem__` 메서드가 `TensorNode`가 아닌 `Tensor`에 구현된 이유는, 연산 그래프의 변형이 주로 `Tensor` 레벨에서 이루어지며, 이를 통해 자동미분 및 그래디언트 계산이 올바르게 수행될 수 있도록 보장하기 위해서임.

1.4 Tensor와 TensorNode의 차이

• Tensor는 실제 데이터를 담고 있는 객체이며, 텐서 연산을 수행하는 기본 단위임. 반면, TensorNode는 그래디언트 계산을 위한 메타 정보를 관리하고, 연산 그래프에서 각 노드를 나타냄.

• Tensor가 TensorNode를 래핑하는 이유는 연산 그래프를 보다 효율적으로 관리하기 위해서임. TensorNode는 각 텐서의 연산과 그래디언트 계산을 관리하는 역할을 수행하고, Tensor는 이러한 TensorNode를 사용해 실제 연산과 데이터를 관리하는 상위 레벨의 인터페이스를 제공함.

• 이와 같은 구조는 텐서 연산을 보다 유연하게 관리할 수 있게 하며, 특히 자동 미분 엔진에서 매우 중요한 역할을 함. Tensor는 사용자가 접근하는 고수준의 인터페이스를 제공하고, TensorNode는 백엔드에서 연산 그래프를 관리하면서 각 연산에 대한 그래디언트 계산을 처리함. 이렇게 함으로써 연산의 추상화를 강화하고, 복잡한 그래프 연산을 효율적으로 처리할 수 있음.

각각의 Python 문법 요소들이 코드에서 어떻게 사용되었는지 설명해드리겠습니다.

1.5 Python 문법 설명

1.5.1 `kwargs`

- `kwargs`는 가변적인 개수의 키워드 인수를 함수에 전달할 때 사용됨. 코드에서는 `Tensor` 클래스의 생성자에서 `kwargs`를 사용하여 다양한 인수를 처리함. 이 방식으로, 사용자가 전달하는 인수에 따라 다른 초기화 로직을 처리할 수 있음.

```
class Tensor:
    def __init__(self, *args, **kwargs) -> None:
        if 'node' in kwargs:
            self.node = kwargs['node']
        else:
            self.node = TensorNode(*args, **kwargs)
```

  - 여기서 `kwargs`는 `node`라는 키워드를 사용하여, 이미 존재하는 `TensorNode` 객체를 직접 할당할 수 있게 해줌. `node` 키워드가 없는 경우에는 새로운 `TensorNode` 객체를 생성하기 위해 `kwargs`를 `TensorNode` 생성자에 그대로 전달함.

1.5.2 `self`

- `self`는 클래스 내부에서 인스턴스 자신을 참조하기 위해 사용되는 변수임. 코드 전반에서 `self`는 클래스 메서드 내에서 인스턴스 변수를 접근하거나 수정하는 데 사용됨.

```
class TensorNode:
    def __init__(self, arr: float | ndarray | TensorNode,
requires_grad: bool = False, is_leaf: bool = True, grad_fn:
Optional[GradFn] = None) -> None:
        self.arr = _ndfy(arr).copy()
        self.requires_grad = requires_grad
        self.is_leaf = is_leaf
        self.grad_fn = grad_fn
        self.grad: Optional[ndarray] = None
        self.grad_cnt = 0
```

- 이 코드에서 `self`는 `TensorNode` 클래스의 각 인스턴스가 고유하게 가지는 `arr`, `requires_grad`, `is_leaf`, `grad_fn`, `grad`, `grad_cnt` 등의 속성에 접근하고 이들을 초기화하는 데 사용됨.

1.5.3 `property`, `property.setter`

- `property` 데코레이터는 클래스의 속성을 메서드처럼 처리할 수 있도록 함. `getter`, `setter`는 속성의 값을 가져오거나 설정할 때 호출되는 메서드를 정의할 수 있게 해줌.

```
class Tensor:
    arr_ = property(lambda self: self.node.arr)
    @arr_.setter
    def arr(self, value: ndarray) -> None:
        self.node.arr = value
```

  - 여기서 `property`는 `arr_` 속성에 대해 정의되었으며, 이를 통해 `arr_` 속성을 접근할 때는 `self.node.arr` 값을 반환함. 또한, `@arr_.setter` 데코레이터를 사용하여 `arr_` 속성에 새로운 값을 설정할 때 `self.node.arr` 값을 업데이트하도록 함. 이를 통해, `Tensor` 객체에서 `arr` 속성에 접근하고 설정하는 방식을 간결하고 직관적으로 관리할 수 있음.

1.5.4 `TypeVar`, `ParamSpec`, `Concatenate`

- 이들은 Python의 `typing` 모듈에서 제공하는 제네릭 타입 힌트임. 이들은 함수나 클래스에서 다양한 타입을 일반화하고 추상화하는 데 사용됨.

```
_P = ParamSpec("_P")
_T = TypeVar("_T")
```

  - 코드에서 `TypeVar`와 `ParamSpec`은 일반화된 함수 시그니처를 정의하기 위해 사용됨. `TypeVar`는 다양한 타입에 대해 함수나 클래스가 작동할 수 있도록 타입 매개변수를 정의하고, `ParamSpec`은 함수의 매개변수 목록을 추상화하는 데 사용됨. 이는 주로 데코레이터나 제네릭 클래스를 정의할 때 유용함.

```
@staticmethod
def _assert_not_leaf(method: Callable[Concatenate[TensorNode,
```

```
_P], _T]) -> Callable[Concatenate[TensorNode, _P], _T]:
    def new_f(self: TensorNode, *args: _P.args, **kwargs:
_P.kwargs) -> _T:
        if self.requires_grad:
            assert not self.is_leaf
        return method(self, *args, **kwargs)
    return new_f
```

- 위의 코드에서 `Concatenate`는 함수 매개변수의 앞에 특정 타입(`TensorNode`)을 추가하는 역할을 함. 이로 인해 `_assert_not_leaf` 데코레이터는 모든 메서드에서 첫 번째 매개변수로 `TensorNode` 타입을 받도록 보장할 수 있음.

---

1. 2. autograd 관련 내용

1.2.1 GradFn의 작동 원리

1.2.1.1 GradFn 클래스 해석: 모든 줄을 한 줄씩 해석하세요.

```
class GradFn:
    def __init__(self, *parents: TensorNode) -> None:
```

- `GradFn` 클래스의 생성자. 이 클래스는 기본적인 그래디언트 함수로, 역전파를 처리하는 기본적인 역할을 함. 입력으로 부모 `TensorNode`들을 받음.

```
        self.parents = parents
```

- 부모 노드들을 `self.parents`에 저장함. 역전파 중에 이 부모 노드들의 그래디언트를 계산할 때 사용됨.

```
    def __call__(self, child: TensorNode) -> None:
        raise NotImplementedError
```

- 이 메서드는 자식 노드(`child`)에서 호출되며, 그래디언트 계산을 수행함. 그러나 `GradFn` 자체는 추상 클래스이기 때문에 이 메서드는 구현되어 있지 않으며, 서브클래스에서 구현해야 함.

2.1.2 f_d가 staticmethod인 경우와 그렇지 않은 경우의 차이

- `staticmethod`인 경우 인스턴스나 클래스의 상태를 참조하지 않으며, 그렇지 않은 경우 인스턴스의 상태를 참조할 수 있음.

1.2.2 GradFn 클래스 및 서브클래스 해석

1.2.2.1 AddGradFn, MulGradFn, DivGradFn, MatMulGradFn의 f_d 해석

1. **AddGradFn**:

```
class AddGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad += child.grad
        self.parents[1].grad += child.grad
```

   - `AddGradFn`은 덧셈 연산에 대한 그래디언트를 계산함. 덧셈 연산의 그래디언트는 입력된 두 텐서 모두에 동일한 값이므로, `child.grad`를 부모 노드들에 더해줌.

2. **MulGradFn**:

```
class MulGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad += child.grad *
self.parents[1].arr
        self.parents[1].grad += child.grad *
self.parents[0].arr
```

   - MulGradFn은 곱셈 연산에 대한 그래디언트를 계산함. 첫 번째 부모의 그래디언트는 두 번째 부모의 값을 곱한 child.grad이고, 두 번째 부모의 그래디언트는 첫 번째 부모의 값을 곱한 child.grad임. 곱셈 연산의 미분 결과는 다른 피연산자에 대해 상호작용하며 계산됨. 이 과정에서 두 피연산자가 서로에게 영향을 주기 때문에, 각각의 그래디언트를 구할 때 상대 피연산자의 값이 사용됨.

3. **DivGradFn**:

```
class DivGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad += child.grad /
```

```
self.parents[1].arr
        self.parents[1].grad -= child.grad *
self.parents[0].arr / self.parents[1].arr**2
```

- `DivGradFn`은 나눗셈 연산에 대한 그래디언트를 계산함. 첫 번째 부모의 그래디언트는 두 번째 부모로 나눈 `child.grad`이고, 두 번째 부모의 그래디언트는 첫 번째 부모와 `child.grad`를 곱하고, 그 값을 두 번째 부모의 제곱으로 나눈 값을 빼는 것임.

4. **MatMulGradFn**:

```
class MatMulGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad += np.matmul(child.grad,
self.parents[1].arr.T)
        self.parents[1].grad +=
np.matmul(self.parents[0].arr.T, child.grad)
```

- `MatMulGradFn`은 행렬 곱셈에 대한 그래디언트를 계산함. 첫 번째 부모의 그래디언트는 두 번째 부모의 전치행렬과 `child.grad`의 행렬 곱으로 계산되며, 두 번째 부모의 그래디언트는 첫 번째 부모의 전치행렬과 `child.grad`의 행렬 곱으로 계산됨.

1.2.2.2 편미분에 대한 고찰

- 편미분은 다변수 함수의 특정 변수에 대한 미분을 의미함.

1.2.2.3 RSubGradFn, RDivGradFn, RPowGradFn, RMatmulGradFn의 f_d가 없는 이유

- 역방향 연산의 특성상 직접적인 그래디언트 계산이 필요하지 않을 수 있음.

1.2.3 상속 및 기타 연산 해석

1.2.3.1 GetitemGradFn, SetitemGradFn, SetitemTensorGradFn 해석

1. **GetitemGradFn**:

```
class GetitemGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad[self.key] += child.grad
```

- `GetitemGradFn`은 `TensorNode`에서 인덱싱(`getitem`) 연산에 대한 그래디언트를 계산함. 특정 `key`에 해당하는 부분에 `child.grad`를 더해줌.

2. **SetitemGradFn**:

```
class SetitemGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad[self.key] += child.grad
```

- `SetitemGradFn`은 `TensorNode`에서 설정(`setitem`) 연산에 대한 그래디언트를 계산함. 특정 `key`에 해당하는 부분에 `child.grad`를 더해줌.

3. **SetitemTensorGradFn**:

```
class SetitemTensorGradFn(GradFn):
    def __call__(self, child: TensorNode) -> None:
        self.parents[0].grad[self.key] += child.grad
        self.parents[1].grad += child.grad[self.key]
```

- `SetitemTensorGradFn`은 `TensorNode`에서 텐서를 설정하는 연산에 대한 그래디언트를 계산함. 첫 번째 부모의 특정 `key`에 해당하는 부분과 두 번째 부모의 전체에 대해 `child.grad`를 더해줌.

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

- `__getitem__`(인덱싱)과 `__setitem__`(아이템 설정) 연산도 미분 가능한 연산임.
- 이유:
  - 미분 가능한 연산이란 입력에 대해 작은 변화를 주었을 때, 그 변화가 출력에 어떻게 영향을 미치는지를 수학적으로 표현할 수 있는 연산을 의미함.
  - `__getitem__` 연산은 텐서에서 특정 부분을 추출하는 작업이므로, 그 특정 부분의 값이 출력에 영향을 미친다면 해당 연산은 미분 가능함. 예를 들어, 행렬의 일부 요소

를 선택하는 연산의 경우, 선택된 요소들이 출력에 영향을 미치고, 그 영향은 미분 계산에서 반영될 수 있음.
    - `__setitem__` 연산은 텐서의 특정 위치에 값을 설정하는 작업임. 이 연산은 기존의 값을 새로운 값으로 대체하는 연산으로, 새로운 값이 그래디언트를 계산하는데 영향을 미칠 수 있기 때문에, 이 연산도 미분 가능함.
- **결론**:
    - 따라서 `__getitem__`과 `__setitem__` 연산 모두 그래디언트 계산의 대상이 될 수 있으며, 이러한 연산들이 포함된 연산 그래프에서 역전파 시 올바른 그래디언트가 계산되도록 보장됨.

1.3. nn 모듈 관련 내용

1.3.1 Parameter와 Module의 구조

1.3.1.1 Parameter와 Tensor의 차이

- `Parameter`는 `Tensor`의 서브클래스로, 주로 학습 가능한 파라미터를 나타냄.

1.3.1.2 He Initialization

- He 초기화는 주로 ReLU 활성화 함수를 사용하는 신경망에서 효과적인 가중치 초기화 방법임.

1.3.2 Module 클래스 해석

1.3.2.1 Module의 구조 해석

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

- `Module` 클래스의 생성자임. 신경망의 각 계층이나 모델을 구성하는 기본 단위로, 모든 신경망 계층이 이 클래스를 상속받음.

```
        self._modules = dict()
```

- 이 클래스 인스턴스가 포함하고 있는 다른 모듈들을 저장하기 위한 사전(dictionary)을 초기화함. 서브모듈들을 이 사전에 저장함.

```
        self._parameters = dict()
```

- 신경망 계층에서 학습 가능한 파라미터들을 저장하는 사전을 초기화함. 예를 들어, 가중치(weight)와 편향(bias) 등이 이 사전에 저장됨.

```
        self.training = True
```

- 이 변수는 모듈이 현재 학습 모드(training mode)인지 여부를 나타냄. 기본값은 `True`로 설정됨.

```
    def forward(self, *args, **kwargs) -> Any:
        raise NotImplementedError
```

- 이 메서드는 각 모듈에서 반드시 구현해야 하는 메서드로, 입력 데이터를 처리하고 출력 데이터를 반환하는 역할을 함. `Module` 클래스에서는 추상 메서드로 정의되어 있어, 하위 클래스에서 구현해야 함.

```
    def __call__(self, *args, **kwargs) -> Any:
        return self.forward(*args, **kwargs)
```

- `__call__` 메서드는 모듈 객체를 함수처럼 호출할 수 있게 함. 내부적으로는 `forward` 메서드를 호출하여 입력 데이터를 처리함.

```
    def parameters(self) -> list:
        return list(self._parameters.values())
```

- 이 메서드는 모듈에 포함된 학습 가능한 파라미터들을 리스트로 반환함. 파라미터는 `_parameters` 사전에서 가져옴.

```
    def add_module(self, name: str, module: Module) -> None:
        self._modules[name] = module
```

- 이 메서드는 새로운 서브모듈을 현재 모듈에 추가함. 서브모듈은 `_modules` 사전에 이름(`name`)과 함께 저장됨.

```
    def train(self, mode: bool = True) -> None:
        self.training = mode
        for module in self._modules.values():
            module.train(mode)
```

- 이 메서드는 모듈을 학습 모드(`training=True`) 또는 평가 모드(`training=False`)로 전환함. 또한, 이 모듈에 포함된 모든 서브모듈들도 동일한 모드로 설정됨.

```
    def eval(self) -> None:
        self.train(False)
```

- 이 메서드는 모듈을 평가 모드로 설정함. `train(False)`를 호출하여 `training` 변수를 `False`로 설정함.

1.3.2.2 Sequential 클래스 해석

`Sequential` 클래스는 `Module` 클래스를 상속받아 여러 계층을 순차적으로 실행할 수 있는 컨테이너를 제공함.

```
class Sequential(Module):
    def __init__(self, *modules: Module) -> None:
        super().__init__()
        for idx, module in enumerate(modules):
            self.add_module(str(idx), module)
```

- `Sequential` 클래스의 생성자임. `Module` 클래스를 상속받아 초기화(`super().__init__()`)하며, 입력받은 여러 `Module` 객체들을 순차적으로 저장함. 각 모듈은 `_modules` 사전에 인덱스(`idx`)를 키로 하여 저장됨.

```
    def forward(self, *args, **kwargs) -> Any:
        for module in self._modules.values():
            args = module(*args, **kwargs)
        return args
```

- `forward` 메서드는 입력 데이터를 순차적으로 각 모듈을 통해 전달하면서 처리함. 마지막 모듈의 출력을 반환함. 각 모듈은 입력 데이터를 처리하여 다음 모듈로 넘김.

```
    def __call__(self, *args, **kwargs) -> Any:
        return self.forward(*args, **kwargs)
```

- `__call__` 메서드는 `Sequential` 객체를 함수처럼 호출할 수 있게 함. 내부적으로 `forward` 메서드를 호출하여 입력 데이터를 처리함.

1.3.3 활성화 함수 및 손실 함수

1.3.3.1 ReLU, Sigmoid, Tanh, CrossEntropyLoss가 Module로 존재하는 이유

- 이 함수들이 `Module`로 존재하는 이유는 각각의 함수가 독립적으로 상태를 관리하거나, 다른 모듈과 동일한 인터페이스를 제공할 필요가 있기 때문임.

1.4. optim 모듈 - SGD 클래스 해석

1.4.1 SGD 클래스의 모든 줄 해석

```
class SGD:
    def __init__(self, params: list, lr: float = 0.01) -> None:
```

- `SGD` 클래스의 생성자임. 이 클래스는 확률적 경사하강법(Stochastic Gradient Descent, SGD)을 구현함. `params`는 업데이트할 모델 파라미터들의 리스트이며, `lr`은 학습률(learning rate)로, 기본값은 0.01로 설정되어 있음.

```
        self.params = params
```

- 전달받은 `params` 리스트를 `self.params`에 저장함. 이 리스트에는 모델의 학습 가능한 파라미터들이 포함됨.

```
        self.lr = lr
```

- 전달받은 학습률 `lr`을 `self.lr`에 저장함. 학습률은 각 파라미터를 업데이트할 때 그래디언트에 곱해지는 값임.

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

- `step` 메서드는 각 학습 파라미터를 업데이트하는 역할을 함. 이 메서드는 일반적으로 역전파(backpropagation)가 끝난 후 호출됨.

```
        for param in self.params:
```

- `self.params`에 포함된 모든 파라미터에 대해 루프를 시작함. 각 파라미터를 하나씩 순회하며 업데이트함.

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

- 각 파라미터의 배열(`arr`)에서 학습률(`lr`)과 해당 파라미터의 그래디언트(`grad`)를 곱한 값을 뺌. 이 식은 기본적인 경사하강법 업데이트 규칙임. 즉, 파라미터를 그래디언트의 반대 방향으로 조금씩 이동시켜 손실을 줄임.

1.5. functions.py 분석

1.5.1 sigmoid_naive와 sigmoid의 차이

- `sigmoid_naive`는 기본적인 시그모이드 함수 구현으로, 단순한 수학적 계산을 수행함. 반면, `sigmoid`는 최적화된 버전으로 더 효율적이거나 추가적인 기능이 있을 수 있음.

1.5.2 log와 LogGradFn

- `log` 함수는 자연 로그를 계산하며, `LogGradFn`은 `log` 함수의 역전파(gradient) 계산을 담당함.

1.5.3 sum과 SumGradFn

- `sum` 함수는 배열의 요소를 모두 더하며, `SumGradFn`은 이 연산의 역전파를 담당함.

1.5.4 relu와 ReLUGradFn

- `relu`는 Rectified Linear Unit 함수로, 입력 값이 0보다 작으면 0을, 그렇지 않으면 그대로 반환함. `ReLUGradFn`은 이 함수의 역전파를 담당함.

1.5.5 repeat와 RepeatGradFn

- `repeat` 함수는 배열을 반복하며, `RepeatGradFn`은 이 연산의 역전파를 담당함.

1.6. main.py 분석 및 설명

1.6.1 f1_score에 대한 간단한 설명

- F1 스코어는 조화 평균을 사용하여 이진 분류 모델의 성능을 평가하는 지표임. 정확도와 재현율의 조화 평균으로 계산됨.

1.6.2 MNIST dataset에 대한 간단한 설명

- MNIST 데이터셋은 0부터 9까지의 숫자 손글씨 이미지로 구성된 데이터셋으로, 컴퓨터 비전 및 패턴 인식 분야에서 널리 사용됨.

1.6.3 model, criterion, optimizer의 선언

- 모델은 신경망의 구조를 정의하고, `criterion`은 손실 함수를, `optimizer`는 모델의 파라미터를 업데이트하는 최적화 알고리즘을 선언함.

1.6.4 학습 루프

- 학습 루프는 데이터셋을 반복적으로 처리하여 모델을 학습시키는 과정임. 각 배치에서 모델이 예측을 수행하고, 손실 함수를 통해 예측 결과와 실제 값 간의 차이를 계산함. 그런 다음 역전파를 통해 그래디언트를 계산하고, 이를 사용해 모델의 파라미터를 업데이트함.

1.6.5 optimizer.zero_grad() 필요한 이유

- `optimizer.zero_grad()`는 이전 배치의 그래디언트가 다음 배치에 누적되지 않도록 매 배치마다 그래디언트를 초기화함. 이를 통해 각 배치에서 독립적인 그래디언트 계산이 가능해짐.

1.6.6 logits = model(x)

- 모델에 입력 `x`를 전달하여 예측 결과인 `logits`을 얻음. `logits`은 활성화 함수가 적용되기 전의 원시 출력임.

1.6.7 loss = criterion(logits, y)

- `criterion`을 사용해 예측된 `logits`과 실제 값 `y` 간의 손실을 계산함. 손실 함수는 예측의 정확성을 평가하는 지표로 사용됨.

1.6.8 criterion이 callable한 이유

- `criterion`은 함수처럼 호출할 수 있음. 손실 함수는 일반적으로 클래스로 구현되지만, `__call__` 메서드를 통해 함수처럼 호출 가능하게 되어 있음. 이를 통해 `criterion(logits, y)`와 같은 형태로 사용 가능함.

1.6.9 loss의 shape

- `loss`는 일반적으로 스칼라 값임. 이는 배치에 대한 평균 손실 또는 합계로 나타나며, 모델이 얼마나 잘 학습되고 있는지를 나타냄.

1.6.10 loss.backward()

- `loss.backward()`는 역전파를 수행하여 손실에 대한 각 파라미터의 그래디언트를 계산함. 이 과정에서 연산 그래프를 따라 각 연산에 대한 미분이 자동으로 계산됨.

1.6.11 optimizer.step()

- `optimizer.step()`은 계산된 그래디언트를 사용하여 모델의 파라미터를 업데이트함. 이 과정에서 파라미터가 손실을 최소화하는 방향으로 조정됨.

1.6.12 macro, micro = val(model, x_val, y_val)

- 이 코드는 검증 데이터셋 `x_val`과 `y_val`을 사용해 모델의 성능을 평가하는 함수임. 여기서 `macro`와 `micro`는 F1 스코어의 매크로 평균과 마이크로 평균을 의미함.

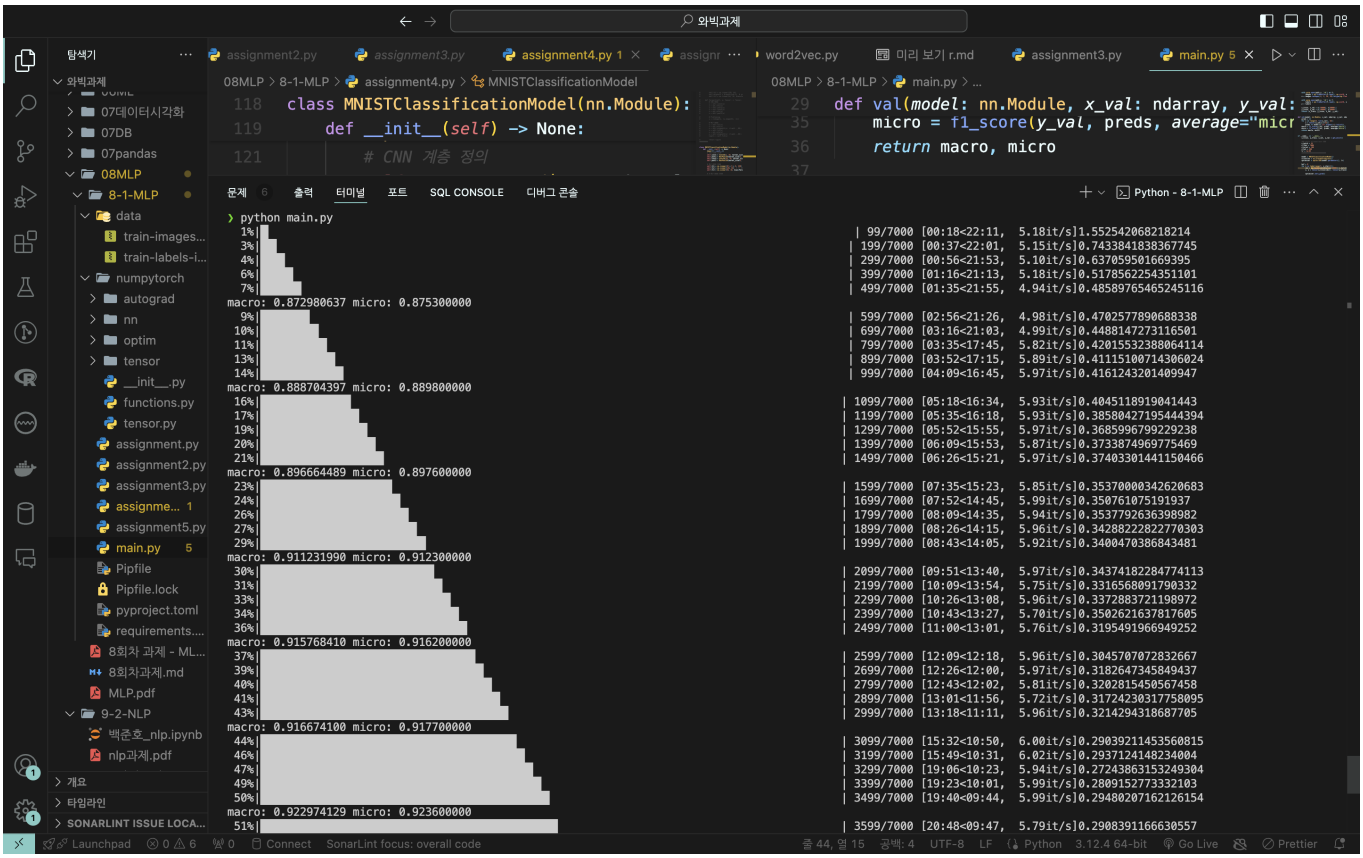
1.6.13 이중 모델 파라미터가 실제로 업데이트되는 건 언제일까요?

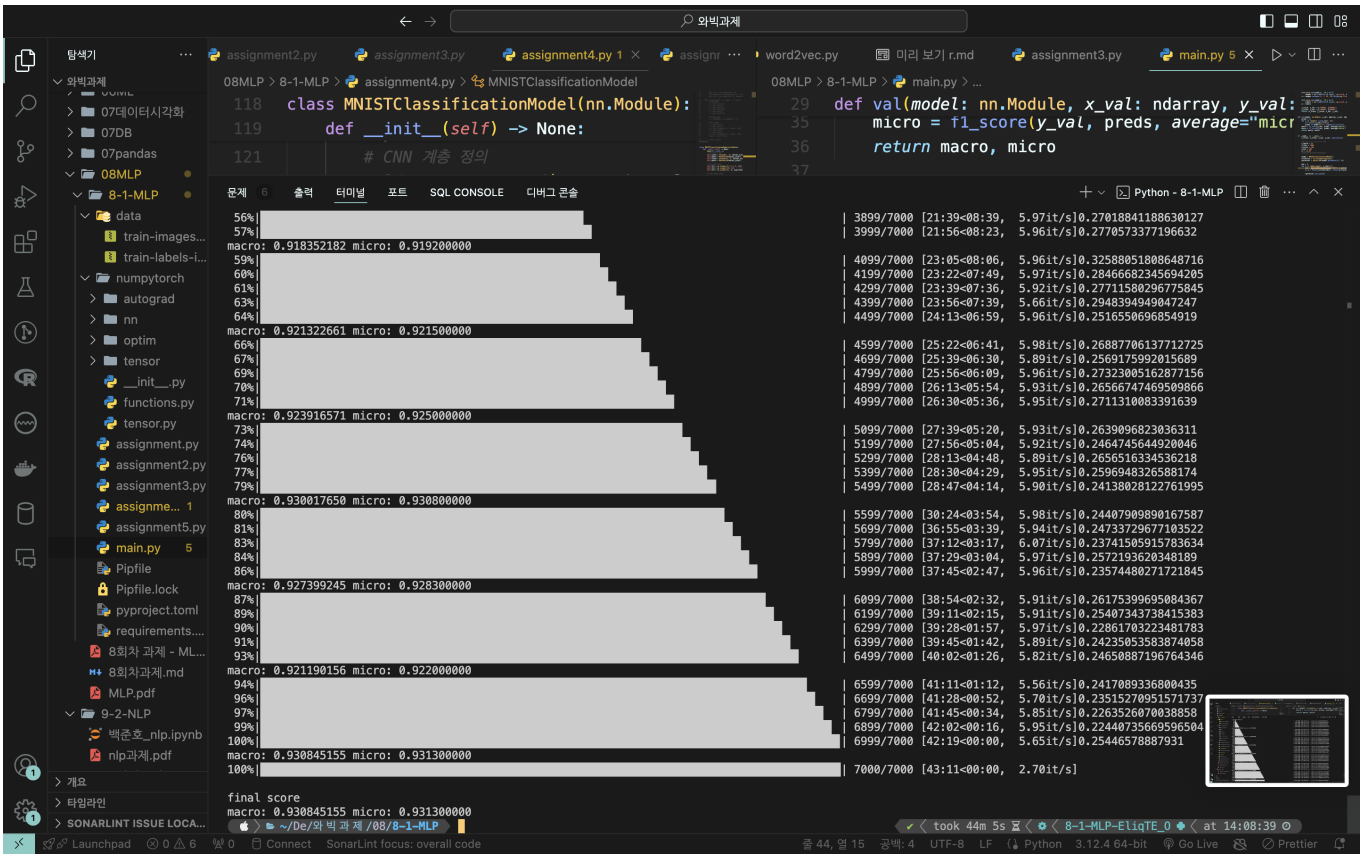
- 모델의 파라미터는 `optimizer.step()`이 호출되는 순간에 업데이트됨. 이 메서드는 `loss.backward()`를 통해 계산된 그래디언트를 사용하여 파라미터를 조정함.

2번 과제

```
n_batch = 32
n_iter = 7000
n_print = 100
n_val = 500
lr = 1e-03
```

main 설정값


assignment4.py
class MNISTClassificationModel(nn.Module):
def __init__(self) -> None:
# CNN 계층 정의
def val(model: nn.Module, x_val: ndarray, y_val:
micro = f1_score(y_val, preds, average="micr
return macro, micro
python main.py
macro: 0.872980637 micro: 0.875300000
macro: 0.888704397 micro: 0.889800000
macro: 0.896664489 micro: 0.897600000
macro: 0.911231990 micro: 0.912300000
macro: 0.915768410 micro: 0.916200000
macro: 0.916674100 micro: 0.917700000
macro: 0.922974129 micro: 0.923600000


assignment4.py
class MNISTClassificationModel(nn.Module):
def __init__(self) -> None:
# CNN 계층 정의
def val(model: nn.Module, x_val: ndarray, y_val:
micro = f1_score(y_val, preds, average="micr
return macro, micro
macro: 0.918352182 micro: 0.919200000
macro: 0.921322661 micro: 0.921500000
macro: 0.923916571 micro: 0.925000000
macro: 0.930017650 micro: 0.930800000
macro: 0.927399245 micro: 0.928300000
macro: 0.921190156 micro: 0.922000000
macro: 0.930845155 micro: 0.931300000
final score
macro: 0.930845155 micro: 0.931300000

결과값