report

8-1-MLP

1. 자동미분엔진 이해하기

1. Tensor

- `Tensor` , `TensorNode`**의 각 attribute이 가지는 의미 (어떻게 사용되는지)**
  - `TensorNode.arr`: 실제 데이터를 저장하는 배열입니다.
  - `TensorNode.requires_grad`: 이 노드의 값에 대한 미분값이 필요하다는 것을 나타냅니다. 역전파 시 그래디언트를 계산할지 여부를 결정합니다.
  - `TensorNode.is_leaf`: 이 노드가 연산 그래프의 말단 노드인지 여부를 나타냅니다. 말단 노드인 경우, `requires_grad`가 True일 때 그래디언트를 계산합니다.
    - `requires_grad`**와의 관계**: `is_leaf`와 `requires_grad`는 연관성이 있습니다. 말단 노드에서 `requires_grad`가 True이면, 그래디언트를 계산해야 합니다.
    - **(**`is_leaf`, `requires_grad`**)의 가능한 조합**: 4가지 (True, True), (True, False), (False, True), (False, False)가 가능합니다.
  - `TensorNode.grad_fn`: 이 노드를 생성한 연산을 나타냅니다. 연산 그래프에서 역전파 시 사용됩니다.

- `TensorNode.grad`: 이 노드에 대해 계산된 그래디언트 값을 저장합니다.
- `TensorNode.grad_cnt`: 현재까지 역전파를 수행한 횟수를 추적합니다.
- `TensorNode.backward`:역전파를 수행하는 메서드로, 그래디언트를 계산하여 연산 그래프의 이전 노드로 전달합니다.
- `TensorNode._create_new_tensornode`: `TensorNode._create_new_tensornode` 메서드는 `TensorNode` 클래스의 인스턴스를 생성하는 역할을 합니다.

```
@staticmethod
def _create_new_tensornode(tensor, requires_grad=False):
    node = TensorNode(tensor, requires_grad=requires_gra
d)
    tensor.grad_fn = node
    return node
```

1. `@staticmethod`
   - `@staticmethod`는 이 메서드가 클래스의 인스턴스나 클래스 자체에 접근하지 않는 정적 메서드임을 나타냅니다. 즉, `self`나 `cls`를 사용하지 않습니다. 인스턴스나 클래스의 상태와 무관하게 작동하는 메서드를 정의할 때 사용합니다.
2. `def _create_new_tensornode(tensor, requires_grad=False):`
   - 이 메서드는 `tensor`와 `requires_grad`라는 두 개의 매개변수를 받습니다. `tensor`는 생성할 `TensorNode`의 기초가 될 텐서입니다. `requires_grad`는 기본적으로 `False`로 설정되며, 이 텐서가 그래디언트 계산을 필요로 하는지 여부를 나타냅니다.
3. `node = TensorNode(tensor, requires_grad=requires_grad)`
   - `TensorNode` 클래스의 새로운 인스턴스를 생성합니다. 생성된 인스턴스는 `node`에 할당되며, 생성자에 `tensor`와 `requires_grad`를 인자로 전달합니다. 이로써 주어진 텐서를 기반으로 하는 `TensorNode` 객체가 만들어집니다.
4. `tensor.grad_fn = node`
   - `tensor`의 `grad_fn` 속성을 `node`로 설정합니다. 이는 `tensor`가 어떤 연산에 의해 생성되었는지를 추적하기 위한 것입니다. `grad_fn` 속성은 주로 자동 미분 과정에서 사용됩니다. 이 설정은 텐서가 `TensorNode`와 연결되어 있고, 필요 시 역전파 과정에서 이 노드를 통해 그래디언트를 계산할 수 있음을 의미합니다.
5. `return node`

- 마지막으로 생성된 `TensorNode` 객체 `node`를 반환합니다. 이 객체는 주어진 텐서에 대한 그래디언트 계산과 관련된 정보를 포함하게 됩니다.

요약하면, `TensorNode._create_new_tensornode` 메서드는 주어진 텐서를 기반으로 `TensorNode` 객체를 생성하고, 해당 텐서의 `grad_fn` 속성에 이 객체를 연결한 뒤, `TensorNode` 객체를 반환합니다. 이 과정은 자동 미분 체계를 구성하는 데 중요한 역할을 합니다.

- `TensorNode._operation` : `TensorNode` 클래스에서 텐서 연산을 처리하는 데 사용되는 메서드입니다.
- `TensorNode._assert_not_leaf`:현재 노드가 말단 노드가 아님을 확인하는 메서드입니다. 말단 노드에서 호출되면 오류를 발생시킵니다.
- `Tensor.setitem` : `TensorNode`가 아닌 `Tensor`에 구현된 이유는, 배열 데이터의 일부를 설정하는 연산이 `Tensor` 객체에서 직접적으로 이루어지기 때문입니다.

- `Tensor`**와** `TensorNode`**의 차이와 분리된 이유**: `Tensor`는 데이터와 연산을 포함하는 고수준 객체이고, `TensorNode`는 연산 그래프의 노드로써 역할을 합니다. 이 둘을 분리함으로써 메모리 관리와 연산 효율성을 향상시킬 수 있습니다.
- **Python 문법**
  - `kwargs`: 키워드 인자를 받는 파라미터로, 함수에 여러 개의 인자를 딕셔너리 형태로 전달할 수 있게 합니다.
  - `self`**는 Python 키워드일까**: `self`는 키워드가 아니며, 관례적으로 클래스 메서드에서 인스턴스 자신을 가리키기 위해 사용되는 첫 번째 매개변수입니다.
  - `property`, `property.setter`
    - **getter와 setter의 개념**: 객체의 속성 값을 읽거나 설정하는 방법을 제공합니다. getter는 속성 값을 반환하고, setter는 속성 값을 설정합니다.
    - `class property`**의 작동 원리**: `@property` 데코레이터를 사용하여 클래스 속성을 정의하면, 이 속성에 접근할 때 getter 메서드가 호출됩니다.
    - `getter`, `setter`, `deleter`**메소드의 동작**: getter는 값을 반환하고, setter는 값을 받아 설정하며, deleter는 속성을 삭제합니다. 이들은 데코레이터로 사용할 수 있습니다.
  - **typing**
    - `TypeVar`, `ParamSpec`, `Concatenate`:
      - **제네릭의 개념**: 특정 타입에 구애받지 않고 여러 타입을 다룰 수 있는 코드를 작성할 수 있도록 합니다.

- `TypeVar`: 제네릭 타입을 정의할 때 사용됩니다.
- `ParamSpec`: 함수의 매개변수 유형을 정의하는 데 사용됩니다.
- `Concatenate`: 매개변수를 결합하여 새로운 함수 서명을 정의하는 데 사용됩니다.

2. autograd

- `GradFn`**의 작동 원리**: `GradFn` 클래스는 미분 연산을 정의하고, 역전파 중에 해당 연산의 그래디언트를 계산하는 역할을 합니다.
  - `class GradFn`**해석**:

```
class GradFn:
    def __init__(self, *args):
        self.inputs = args

    def backward(self, grad_output):
        raise NotImplementedError()
```

1. `class GradFn`

   GradFn이라는 클래스를 정의합니다. 이 클래스는 자동 미분의 각 연산에 대한 그래디언트 계산을 담당합니다.

2. `def __init__(self, *args)`

   GradFn 클래스의 생성자입니다. 연산의 입력들을 `args`로 받습니다.

3. `self.inputs = args`

   입력된 인자를 저장합니다. 이 값들은 나중에 역전파를 수행할 때 사용됩니다.

4. `def backward(self, grad_output)`

   `backward` 메서드는 역전파를 수행하며, 자식 클래스에서 오버라이딩하여 각 연산에 맞는 그래디언트 계산을 수행합니다. 여기서는 `grad_output`을 입력으로 받아야 합니다.

  - `f_d`**가 staticmethod인 경우와 그렇지 않은 경우의 차이**: staticmethod는 클래스 인스턴스와 관련 없이 호출되며, 인스턴스를 통해 호출될 필요가 없습니다.
  - `AddGradFn`, `MulGradFn`, `DivGradFn`, `MatMulGradFn`**의** `f_d` **해석**: 각각의 함수에서 f_d는 미분 계산의 방법을 정의합니다.

    - **편미분에 대한 설명**: 다변수 함수의 각 변수에 대해 하나씩 미분한 값을 의미합니다.

- `RSubGradFn`, `RDivGradFn`, `RPowGradFn`, `RMatmulGradFn`**이** `f_d`**가 없는 이유**: 역방향 연산이 이미 정의된 연산과 대칭적인 관계를 갖기 때문에, 별도의 f_d가 필요하지 않습니다.

- `GetitemGradFn`, `SetitemGradFn`, `SetitemTensorGradFn`**해석**: 각각 `getitem`, `setitem` 연산에 대한 미분 방법을 정의합니다.

    - **getitem, __setitem__도 미분가능한 연산인가?**: 네, 미분 가능하며, 이를 통해 텐서의 특정 요소에 대한 연산이 미분됩니다.

3. nn

- `Parameter`, `Module`**의 구조**:

  PyTorch의 `nn` 모듈은 신경망을 정의하고 학습할 때 유용한 클래스들을 제공합니다. `Parameter`와 `Module`은 이러한 클래스 중에서 핵심적인 역할을 합니다.

    - **`Parameter`**:

        - `Parameter`는 신경망에서 학습 가능한 매개변수를 나타냅니다. `torch.Tensor`를 상속하여 만들어지며, 자동으로 신경망의 매개변수로 등록됩니다.

    - **`Module`**:

        - `Module`은 신경망의 레이어 또는 전체 모델을 나타냅니다. `nn.Module`을 상속하여 신경망의 각 레이어를 구현할 수 있습니다. `Module`은 내부적으로 `Parameter` 객체를 관리하며, 다양한 신경망 연산을 수행합니다.

    - **Tensor와 Parameter의 차이**: `Parameter`는 `requires_grad`가 기본적으로 True이며, 신경망의 가중치로 사용됩니다.

    - **He initialization**: **He Initialization**(He 초기화)는 Xavier 초기화의 변형으로, ReLU 활성화 함수와 같은 비선형 활성화 함수를 사용할 때 적합한 가중치 초기화 방법입니다. He 초기화는 가중치를 정규분포로 초기화하는데, 표준편차를 `sqrt(2/n)`로 설정합니다. 여기서 `n`은 입력 노드의 수입니다. 이는 ReLU와 같은 활성화 함수가 입력의 절반을 0으로 만들기 때문에, 가중치를 더 크게 초기화하여 신호의 소멸을 방지합니다.

    - `class Module` **해석**:

```
class Module:
    def __init__(self):
        self._parameters = {}
```

```
        self._modules = {}
        self.training = True
```

1. `class Module:`

   `Module` 클래스를 정의합니다. 이 클래스는 신경망의 레이어 또는 전체 모델을 표현하는 기본 클래스입니다.

2. `def __init__(self):`

   `Module` 클래스의 생성자입니다.

3. `self._parameters = {}`

   신경망의 매개변수(즉, `Parameter` 객체들)를 저장하기 위한 딕셔너리를 초기화합니다.

4. `self._modules = {}`

   이 모듈이 포함하는 하위 모듈들을 저장하기 위한 딕셔너리를 초기화합니다. 예를 들어, 여러 레이어로 구성된 모델을 정의할 때, 각 레이어를 하위 모듈로 관리할 수 있습니다.

5. `self.training = True`

   모델이 현재 학습 모드인지 여부를 나타냅니다. 모델이 학습 중일 때는 `True`로 설정되고, 평가 모드에서는 `False`로 설정됩니다.

- `class Sequential` **해석**:

```
class Sequential(Module):
    def __init__(self, *args):
        super(Sequential, self).__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)
```

1. `class Sequential(Module):`

   `Sequential` 클래스는 `Module`을 상속합니다. 이 클래스는 신경망의 레이어들을 순차적으로 쌓아 구성할 때 사용됩니다.

2. `def __init__(self, *args):`

   `Sequential` 클래스의 생성자입니다. 가변 인자로 여러 모듈을 받을 수 있습니다.

3. `super(Sequential, self).__init__()`

부모 클래스인 `Module`의 생성자를 호출하여, 상속된 속성을 초기화합니다.

4. `for idx, module in enumerate(args):`

   전달된 모듈들을 순차적으로 열거합니다. `idx`는 모듈의 인덱스, `module`은 각각의 모듈 객체입니다.

5. `self.add_module(str(idx), module)`

   각 모듈을 `add_module` 메서드를 통해 `Sequential` 객체에 추가합니다. `add_module`은 모듈을 이름과 함께 등록하며, `str(idx)`는 모듈의 이름이 됩니다.

- **`ReLU`, `Sigmoid`, `Tanh`, `CrossEntropyLoss`가 `Module`로 존재하는 이유**:

이러한 활성화 함수와 손실 함수가 `Module`로 존재하는 이유는 다음과 같습니다:

1. **통일된 인터페이스**:
   - PyTorch에서 신경망 레이어와 손실 함수는 모두 `Module`을 기반으로 구현됩니다. 이는 일관된 인터페이스를 제공하여, 모델 구성과 학습에서 사용이 편리합니다.
2. **자동 등록 및 관리**:
   - `Module` 클래스는 학습 가능한 매개변수(`Parameter`)를 자동으로 추적하고 관리합니다. 활성화 함수와 손실 함수가 `Module`로 존재할 때, 이들과 관련된 매개변수(필요한 경우)가 자동으로 관리됩니다.
3. **쉽고 일관된 학습 모드와 평가 모드 전환**:
   - `Module`로 존재하는 함수는 학습 모드(`train()`)와 평가 모드(`eval()`) 전환 시, 해당 함수들이 일관되게 동작합니다. 이는 특히 배치 정규화와 드롭아웃 같은 경우 중요합니다.
4. **복잡한 연산 포함 가능**:
   - `ReLU`, `Sigmoid`, `Tanh` 등은 단순한 활성화 함수이지만, `CrossEntropyLoss`는 내부적으로 소프트맥스 연산과 로그 연산을 포함한 복잡한 연산을 수행합니다. `Module`로 구현됨으로써 이러한 복잡한 연산도 쉽게 정의할 수 있습니다.

4. optim

- `class SGD` **해석**:

```
import torch
from torch.optim.optimizer import Optimizer

class SGD(Optimizer):
    def __init__(self, params, lr=0.01, momentum=0, dampen
```

```
                    weight_decay=0, nesterov=False):
        defaults = dict(lr=lr, momentum=momentum, dampenin
                        weight_decay=weight_decay, nestero
        super(SGD, self).__init__(params, defaults)
        if nesterov and (momentum <= 0 or dampening != 0):
            raise ValueError("Nesterov momentum requires a
```

PyTorch에서 `SGD`는 확률적 경사 하강법(Stochastic Gradient Descent) 알고리즘을 구현한 클래스입니다. `torch.optim` 모듈에서 제공되며, 모델의 매개변수를 학습하는 데 사용됩니다.

1. `import torch`

   PyTorch 라이브러리를 불러옵니다. 이는 텐서 연산 및 다양한 유틸리티를 제공합니다.

2. `from torch.optim.optimizer import Optimizer`

   PyTorch의 `Optimizer` 클래스를 가져옵니다. 이는 모든 최적화 알고리즘의 기본 클래스입니다.

3. `class SGD(Optimizer):`

   `SGD` 클래스를 정의하며, `Optimizer` 클래스를 상속받습니다. 이는 SGD 알고리즘을 구현합니다.

4. `def __init__(self, params, lr=0.01, momentum=0, dampening=0, weight_decay=0, nesterov=False):`

   `SGD` 클래스의 생성자입니다. 학습할 매개변수와 함께 학습률(`lr`), 모멘텀(`momentum`), 감쇠(`dampening`), 가중치 감쇠(`weight_decay`), Nesterov 모멘텀 사용 여부(`nesterov`)를 초기화합니다.

5. `defaults = dict(lr=lr, momentum=momentum, dampening=dampening, weight_decay=weight_decay, nesterov=nesterov)`

   전달받은 하이퍼파라미터들을 딕셔너리 형태로 저장하여, 이후 부모 클래스인 `Optimizer`에 전달합니다.

6. `super(SGD, self).__init__(params, defaults)`

   부모 클래스인 `Optimizer`의 생성자를 호출하여, 매개변수와 기본 설정값을 초기화합니다.

7. `if nesterov and (momentum <= 0 or dampening != 0):`

   Nesterov 모멘텀을 사용할 경우, 모멘텀 값이 0보다 크고 감쇠가 0이어야 한다는 조건을 확인합니다.

8. `raise ValueError("Nesterov momentum requires a momentum and zero dampening")`

   위의 조건이 충족되지 않으면 오류를 발생시켜, Nesterov 모멘텀 사용 시 잘못된 설정이 적용되지 않도록 합니다.

```
    def step(self, closure=None):
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                d_p = p.grad.data
                if group['weight_decay'] != 0:
                    d_p.add_(group['weight_decay'], p.d
                if group['momentum'] != 0:
                    param_state = self.state[p]
                    if 'momentum_buffer' not in param_s
                        buf = param_state['momentum_buf
                    else:
                        buf = param_state['momentum_buf
                        buf.mul_(group['momentum']).add_
                    if group['nesterov']:
                        d_p = d_p.add(group['momentum']
                    else:
                        d_p = buf

                p.data.add_(-group['lr'], d_p)

        return loss
```

1. `def step(self, closure=None):`
   - 매개변수 업데이트를 수행하는 메서드입니다. 옵셔널로 손실 함수를 인자로 받아, 필요 시 다시 계산할 수 있습니다.
2. `loss = None`
   - 손실 값을 `None`으로 초기화합니다.

3. `if closure is not None:`
    - `closure` 함수가 제공되었는지 확인합니다.
4. `loss = closure()`
    - `closure` 함수가 제공되었다면 이를 호출하여 손실을 계산하고 저장합니다.
5. `for group in self.param_groups:`
    - 모든 매개변수 그룹을 순회하며 최적화합니다. `param_groups`는 매개변수를 포함하는 딕셔너리입니다.
6. `for p in group['params']:`
    - 각 매개변수에 대해 순회합니다.
7. `if p.grad is None:`
    - 매개변수에 대한 그래디언트가 없다면, 해당 매개변수를 건너뜁니다.
8. `d_p = p.grad.data`
    - 매개변수의 그래디언트를 `d_p`에 저장합니다.
9. `if group['weight_decay'] != 0:`
    - `weight_decay` 값이 0이 아니면, `d_p`에 가중치 감쇠를 적용합니다. 이는 가중치 감소(L2 정규화)를 수행합니다.
10. `d_p.add_(group['weight_decay'], p.data)`
    - `d_p`에 가중치 감쇠 값을 더합니다.
11. `if group['momentum'] != 0:`
    - 모멘텀 값이 0이 아니면, 모멘텀을 적용하여 그래디언트를 업데이트합니다.
12. `param_state = self.state[p]`
    - 매개변수의 상태를 저장할 `param_state`를 불러옵니다.
13. `if 'momentum_buffer' not in param_state:`
    - 매개변수 상태에 모멘텀 버퍼가 존재하는지 확인합니다.
14. `buf = param_state['momentum_buffer'] = torch.clone(d_p).detach()`
    - 모멘텀 버퍼가 없으면, 현재의 `d_p`를 복제하여 새로운 버퍼로 저장합니다.
15. `else:`
    - 모멘텀 버퍼가 이미 존재하면, 기존 버퍼를 업데이트합니다.

16. `buf = param_state['momentum_buffer']`
    - 기존 모멘텀 버퍼를 `buf`에 할당합니다.
17. `buf.mul_(group['momentum']).add_(1 - group['dampening'], d_p)`
    - 버퍼에 모멘텀 값을 곱하고, 감쇠 값을 반영하여 `d_p`를 더합니다.
18. `if group['nesterov']:`
    - Nesterov 모멘텀이 활성화된 경우, Nesterov 업데이트를 수행합니다.
19. `d_p = d_p.add(group['momentum'], buf)`
    - 기존 그래디언트에 모멘텀 버퍼를 더하여 업데이트된 `d_p`를 생성합니다.
20. `else:`
    - Nesterov 모멘텀이 아닌 경우, 일반적인 모멘텀 업데이트를 수행합니다.
21. `d_p = buf`
    - `d_p`를 모멘텀 버퍼로 대체합니다.
22. `p.data.add_(-group['lr'], d_p)`
    - 최종적으로 계산된 `d_p`를 학습률(`lr`)과 함께 매개변수 `p`에 반영합니다. 이는 매개변수의 값이 업데이트되는 부분입니다.
23. `return loss`
    - 손실 값을 반환합니다. `closure` 함수가 제공되었다면, 이 값이 계산된 손실 값이 될 수 있습니다.

5. functions.py

- `sigmoid_naive`**와** `sigmoid`**의 차이**: `sigmoid_naive`는 단순한 계산을 수행하고, `sigmoid`는 메모리와 성능 최적화를 고려한 버전입니다.
- `log`**와** `LogGradFn`: `log`는 로그 함수이고, `LogGradFn`은 해당 함수의 그래디언트를 계산합니다.
- `sum`**과** `SumGradFn`: `sum`은 합을 계산하며, `SumGradFn`은 해당 연산의 그래디언트를 계산합니다.
- `relu`**와** `ReLUGradFn`: `relu`는 ReLU 활성화 함수이며, `ReLUGradFn`은 이에 대한 그래디언트를 계산합니다.
- `repeat`**와** `RepeatGradFn`: `repeat`는 텐서를 반복하며, `RepeatGradFn`은 이 연산의 그래디언트를 계산합니다.

6. main.py

- **`f1_score`에 대한 설명**:

  `f1_score`는 분류 문제에서 모델의 성능을 평가하기 위해 사용되는 지표입니다. `f1_score`는 정밀도(Precision)와 재현율(Recall)의 조화 평균을 의미하며, 두 값의 조화 평균이기 때문에 정밀도와 재현율의 균형을 중요시합니다.

  정밀도는 모델이 양성으로 예측한 것 중 실제로 양성인 비율을, 재현율은 전체 실제 양성 중에서 모델이 양성으로 예측한 비율을 나타냅니다. `f1_score`는 불균형한 클래스 분포를 가진 데이터에서 유용합니다.

- **MNIST dataset에 대한 설명**:

  `MNIST`(Modified National Institute of Standards and Technology) 데이터셋은 손으로 쓴 숫자 이미지(0-9)의 모음으로, 70,000개의 흑백 이미지로 구성되어 있습니다. 각 이미지는 28x28 픽셀 크기로, 픽셀 값은 0에서 255 사이의 그레이스케일 값을 가집니다. 이 데이터셋은 이미지 분류 문제에서 가장 널리 사용되는 벤치마크 중 하나로, 많은 머신러닝 알고리즘이 이 데이터셋을 통해 성능을 평가받습니다.

- **`model`, `criterion`, `optimizer`의 선언**:
  - **`model`**:
    - 신경망의 구조를 정의하는 부분입니다. `model`은 `nn.Module`을 상속하여 정의되며, 데이터가 입력되었을 때 어떤 연산을 통해 출력이 계산되는지 결정합니다.
  - **`criterion`**:
    - 손실 함수(loss function)를 정의합니다. 손실 함수는 모델의 출력과 실제 레이블 간의 차이를 측정하며, 모델 학습 중 최소화하려는 목표입니다. 예를 들어, `CrossEntropyLoss`는 다중 클래스 분류 문제에서 자주 사용됩니다.
  - **`optimizer`**:
    - 모델의 매개변수를 업데이트하는 방법을 정의합니다. 예를 들어, `SGD`나 `Adam`과 같은 최적화 알고리즘이 사용됩니다. 옵티마이저는 손실 함수의 그래디언트를 기반으로 모델의 가중치를 업데이트합니다.
- **학습 루프**: 데이터를 반복적으로 모델에 공급하고, 손실을 계산하며, 역전파를 통해 모델 파라미터를 업데이트하는 과정입니다.
  - **`optimizer.zero_grad()`의 이유**: `optimizer.zero_grad()`는 현재 배치에서 계산된 그래디언트를 초기화하는 역할을 합니다. PyTorch에서 `backward()` 함수를 호출하면, 그래디언트가 누적되므로, 이전 배치의 그래디언트가 현재 배치에 영향을 미치지 않도록 하기 위해 초기화가 필요합니다.

- `logits = model(x)` : `model`에 입력 데이터 `x`를 전달하여 로짓(logits)을 계산합니다. 로짓은 출력층의 활성화 함수를 통과하기 전의 모델의 예측값으로, 주로 소프트맥스 활성화 함수와 함께 사용됩니다.
- `loss = criterion(logits, y)`: 손실 함수 `criterion`에 모델의 출력값인 로짓 `logits`와 실제 레이블 `y`를 전달하여 손실을 계산합니다. 이 손실 값은 모델이 얼마나 부정확한지 측정하는 값입니다.
  - `criterion`이 callable한 이유: 손실 함수는 `criterion(logits, y)`처럼 호출되며, 이는 파이썬에서 함수나 객체가 호출 가능한(callable) 이유와 동일합니다. 이는 `criterion` 객체가 `__call__` 메서드를 구현하고 있기 때문입니다. PyTorch의 손실 함수들은 클래스 형태로 구현되어 있지만, 마치 함수처럼 사용할 수 있도록 callable하게 설계되었습니다.
  - `loss`의 shape: `loss`의 shape은 보통 `torch.Size([])`로, 스칼라 값입니다. 이는 미니 배치에서 평균 손실을 반환하기 때문에 단일 값으로 나타납니다.
- `loss.backward()` : 역전파를 수행하여 손실에 대한 각 매개변수의 그래디언트를 계산합니다. 이 과정에서 각 매개변수의 `grad` 속성이 업데이트됩니다.
- `optimizer.step()`: 계산된 그래디언트를 바탕으로 모델의 매개변수를 업데이트합니다. 최적화 알고리즘(SGD, Adam 등)에 따라 모델의 파라미터를 갱신합니다.
- `macro, micro = val(model, x_val, y_val)`: 모델 `model`을 검증 데이터 `x_val`과 `y_val`에 대해 평가하여, 매크로 및 마이크로 F1 스코어를 계산합니다. `macro`는 각 클래스별 F1 스코어의 평균을, `micro`는 전체 평균을 나타냅니다.
- **모델 파라미터가 실제로 업데이트되는 건 언제일까요?**: 모델 파라미터가 실제로 업데이트되는 순간은 `optimizer.step()`이 호출될 때입니다. `loss.backward()`는 각 매개변수의 그래디언트를 계산하는 단계이고, `optimizer.step()`에서 이 그래디언트를 바탕으로 매개변수를 조정합니다.

2. CNN 구현하기

`Conv2d` 클래스

```
class Conv2d(nn.Module):
    def __init__(self, in_channels: int, out_channels: int,
kernel_size: int, stride: int = 1, padding: int = 0) -> Non
e:
        super().__init__()
        # 입력 채널 수, 출력 채널 수, 커널 크기, 스트라이드, 패딩을
```

```
초기화합니다.
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        # 가중치와 편향을 초기화합니다. 가중치는 정규 분포로 초기화하
며, 편향은 0으로 초기화합니다.
        self.weight = Tensor(np.random.randn(out_channels,
in_channels, kernel_size, kernel_size) * np.sqrt(1. / in_ch
annels))
        self.bias = Tensor(np.zeros(out_channels))

    def forward(self, x: Tensor) -> Tensor:
        # 입력 텐서의 배치 크기, 채널 수, 높이, 너비를 추출합니다.
        batch_size, _, height, width = x.shape

        # 출력 텐서의 높이와 너비를 계산합니다.
        out_height = (height + 2 * self.padding - self.kern
el_size) // self.stride + 1
        out_width = (width + 2 * self.padding - self.kernel
_size) // self.stride + 1

        # 입력 텐서에 패딩을 추가합니다.
        x_padded = np.pad(x.arr, ((0,), (0,), (self.paddin
g,), (self.padding,)), mode='constant')

        # 출력 텐서를 초기화합니다.
        output = np.zeros((batch_size, self.out_channels, o
ut_height, out_width))

        # 각 위치에서 필터를 적용합니다.
        for i in range(out_height):
            for j in range(out_width):
                # 필터가 적용될 지역을 추출합니다.
                region = x_padded[:, :, i*self.stride:i*sel
f.stride+self.kernel_size, j*self.stride:j*self.stride+sel
```

```
f.kernel_size]
                # 지역과 필터를 텐서 곱셈하여 결과를 출력 텐서에 저장
합니다.
                output[:, :, i, j] = np.tensordot(region, s
elf.weight.arr, axes=([1, 2, 3], [1, 2, 3])) + self.bias.ar
r

        # NumPy 배열을 Tensor 객체로 변환하여 반환합니다.
        return Tensor(output)
```

`MaxPool2d` 클래스

```
class MaxPool2d(nn.Module):
    def __init__(self, kernel_size: int, stride: int) -> No
ne:
        super().__init__()
        # 풀링 필터의 크기와 스트라이드를 초기화합니다.
        self.kernel_size = kernel_size
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        # 입력 텐서의 배치 크기, 채널 수, 높이, 너비를 추출합니다.
        batch_size, channels, height, width = x.shape

        # 출력 텐서의 높이와 너비를 계산합니다.
        out_height = (height - self.kernel_size) // self.st
ride + 1
        out_width = (width - self.kernel_size) // self.stri
de + 1

        # 출력 텐서를 초기화합니다.
        output = np.zeros((batch_size, channels, out_heigh
t, out_width))

        # 각 위치에서 풀링 필터를 적용합니다.
        for i in range(out_height):
            for j in range(out_width):
                # 필터가 적용될 지역을 추출합니다.
```

```
                region = x.arr[:, :, i*self.stride:i*self.s
tride+self.kernel_size, j*self.stride:j*self.stride+self.ke
rnel_size]
                # 지역에서 최대값을 추출하여 출력 텐서에 저장합니다.
                output[:, :, i, j] = np.max(region, axis=
(2, 3))

        # NumPy 배열을 Tensor 객체로 변환하여 반환합니다.
        return Tensor(output)
```

`MNISTClassificationModel` 클래스

```
class MNISTClassificationModel(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        # 모델의 레이어를 정의합니다.
        # 첫 번째 합성곱 레이어: 입력 채널 1, 출력 채널 16, 커널 크
기 3
        self.conv1 = Conv2d(1, 16, 3)  # 28x28 -> 26x26
        # 최대 풀링 레이어: 커널 크기 2, 스트라이드 2
        self.pool = MaxPool2d(2, 2)    # 26x26 -> 13x13
        # 두 번째 합성곱 레이어: 입력 채널 16, 출력 채널 32, 커널 크
기 3
        self.conv2 = Conv2d(16, 32, 3) # 13x13 -> 11x11
        # 전결합 레이어: 32 * 5 * 5 입력 노드, 120 출력 노드
        self.fc1 = nn.Linear(32 * 5 * 5, 120)
        # 전결합 레이어: 120 입력 노드, 84 출력 노드
        self.fc2 = nn.Linear(120, 84)
        # 전결합 레이어: 84 입력 노드, 10 출력 노드 (클래스 수)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x: Tensor) -> Tensor:
        # 합성곱 레이어와 풀링 레이어를 순차적으로 적용합니다.
        x = self.pool(self.conv1(x))  # 28x28 -> 26x26 -> 1
3x13
        x = self.pool(self.conv2(x))  # 13x13 -> 11x11 -> 5
x5
```

```
        # 텐서를 NumPy 배열로 변환하여 형태를 변경합니다.
        x_np = x.arr
        x_np = x_np.reshape(-1, 32 * 5 * 5)  # 형태 변경 (배치
크기, 32 * 5 * 5)

        # 형태가 변경된 NumPy 배열을 다시 Tensor 객체로 변환합니다.
        x = Tensor(x_np)

        # 전결합 레이어와 활성화 함수 적용
        x = nn.relu(self.fc1(x))  # 첫 번째 전결합 레이어
        x = nn.relu(self.fc2(x))  # 두 번째 전결합 레이어
        x = self.fc3(x)           # 마지막 전결합 레이어

        # 최종 로짓을 반환합니다.
        return x
```

이 코드들은 `Conv2d`와 `MaxPool2d` 레이어를 정의하고, MNIST 데이터셋을 위한 CNN 모델을 구성하는데 필요한 클래스를 구현합니다. 각 클래스의 `forward` 메서드는 입력 데이터에 대해 연산을 수행하고 결과를 반환합니다. 이 구현 방식은 Tensor 객체의 메서드가 제한적일 때 NumPy 배열을 활용하는 방법을 포함하고 있습니다.

최종결과