

파이토치 & 클래스

구준회 권도영 서은서 황영서

CONTENTS

1. 클래스

2. 파이토치 기본 문법

3. 파이토치 모델 구현

4. 파이토치 실습

클래스

구분회

CONTENTS

1. 클래스와 객체

2. 클래스 만들기

3. 클래스 상속

01. 클래스와 객체

클래스 : 제품의 설계도



01. 클래스와 객체

객체 : 설계도로 만든 제품



>> 하나의 **클래스**로부터
여러 개의 **객체**를 생성할 수 있다.

클래스 용어정리

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

클래스 만들기

```
Class Pokemon:  
    def __init__(self, name, types): # 생성자  
        self.name = name  
        self.types = types  
    def say(self): # 메소드  
        print(f"{self.types}타입 포켓몬 {self.name}")
```

```
# 객체  
파이리 = Pokemon("파이리", "불")  
파이리.say()
```

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

클래스 만들기

```
Class Pokemon:
    def __init__(self, name, types) # 생성자
        self.name = name
        self.types = types
    def say(self): # 메소드
        print(f"{self.types}타입 포켓몬 {self.name}")
```

```
# 객체
파이리 = Pokemon("파이리", "불")
파이리.say()
```

➡ Quiz. 이 결과값은?

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

Setter Method

```
class Pokemon:
    def __init__(self, name, types):
        self.name = name
        self.types = types

    def say(self):
        print(f"{self.types}타입 포켓몬 {self.name}")

    # Setter 메소드
    def set_name(self, name):
        self.name = name
```

```
파이리 = Pokemon('파이리', '불')
파이리.set_name("꼬부기")
print(파이리.name, '를 잡았습니다!')
```

꼬부기 를 잡았습니다!

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

02. 클래스 만들기

클래스 소멸자

```
# 소멸자
def __del__(self):
    print('인스턴스를 소멸시킵니다!')

# Setter 메소드
def set_name(self, name):
    self.name = name

파이리 = Pokemon('파이리', '불')
파이리.set_name("꼬부기")
print(파이리.name, '를 잡았습니다!')
del 파이리
파이리.say()
```

꼬부기 를 잡았습니다!
인스턴스를 소멸시킵니다!

```
-----
NameError                                Traceback (most recent call last)
Cell In[10], line 21
    19 print(파이리.name, '를 잡았습니다!')
    20 del 파이리
--> 21 파이리.say()

NameError: name '파이리' is not defined
```

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

클래스 상속

- 정의 : 다른 클래스의 변수와 메소드를 물려받아 사용하는 기법
- 부모와 자식 관계가 존재
- 자식 클래스 : 부모 클래스를 상속받은 클래스

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

클래스 상속

```
class Pokemon:
    def __init__(self, name, tech):
        self.name = name
        self.tech = tech

    def attack(self):
        print(f"{self.name} 이(가) {self.tech}를 사용했습니다!")
```

```
피카츄 = Pokemon('피카츄', '몸통박치기')
피카츄.attack()
```

피카츄 이(가) 몸통박치기를 사용했습니다!

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

클래스 상속

```
In [23]: class Pokemon:
    def __init__(self, name, tech):
        self.name = name
        self.tech = tech

    def attack(self):
        print(f"{self.name} 이(가) {self.tech}를 사용했습니다!")

class Monster(Pokemon):
    def __init__(self, name, tech, level):
        self.name = name
        self.tech = tech
        self.level = level
    def show_info(self):
        print('이름 :', self.name, "/ 레벨 :", self.level)

피카츄 = Monster('피카츄', '몸통박치기', 10)
피카츄.show_info()
피카츄.attack()

이름 : 피카츄 / 레벨 : 10
피카츄 이(가) 몸통박치기를 사용했습니다!
```

자식 클래스에서 부모 클래스의 속성, 메소드를 사용할 수 있지만 반대로는 불가능

| 용어 | 설명 |
|------|------------------|
| 클래스 | 제품의 설계도 |
| 객체 | 설계도로 만든 제품 |
| 속성 | 클래스 안의 변수 |
| 메소드 | 클래스 안의 함수 |
| 생성자 | 객체를 만들 때 실행되는 함수 |
| 인스턴스 | 메모리 내에 살아있는 객체 |

참고문헌

동빈나 유튜브 : <https://www.youtube.com/watch?v=YQhJsWj6ydU&t=370s>

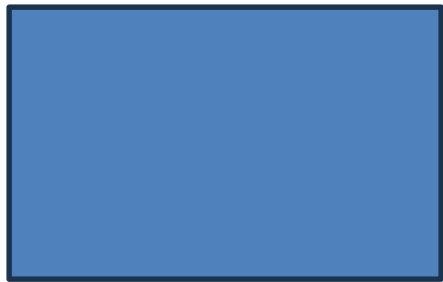
파이토치 기본 문법

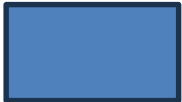
황영서

파이토치란?



- facebook에서 발표한 딥러닝 구현을 위한 파이썬 기반의 오픈소스 머신러닝 라이브러리
- 특징 : 1) Numpy를 대체하면서 GPU를 이용하여 연산 가능
2) Dynamic Computing Graph(Define By Run) 으로 진행



는 PyTorch에서 데이터를 표현하기 위해 사용하는 기본 구조
넘파이의 ndarray와 유사, GPU 를 사용하여 연산 가능

```
import torch
```

```
print(torch.tensor([[1,2],[3,4]]))  
print(torch.tensor([[1,2],[3,4]], device="cuda:0"))    # GPU에 텐서 생성  
print(torch.tensor([[1,2],[3,4]], dtype=torch.float64))
```

```
tensor([[1, 2],  
        [3, 4]])  
tensor([[1, 2],  
        [3, 4]], device='cuda:0')  
tensor([[1., 2.],  
        [3., 4.]], dtype=torch.float64)
```

tensor vs Tensor

- `torch.tensor` (Python function, in `torch.tensor`)
- `torch.Tensor` (Python class, in `torch.Tensor`)

tensor vs Tensor

1. type 의 차이

```
array = np.array([1,2,3])

Tensor = torch.Tensor(array)
tensor = torch.tensor(array)

print(Tensor)
print(tensor)
print(Tensor.dtype)
print(tensor.dtype)
```

output

```
tensor([1., 2., 3.])
tensor([1, 2, 3])
torch.float32
torch.int64
```

→ Tensor()는 type를 Float 고정이며,
tensor()는 입력 데이터에 따라 type이 변함

tensor vs Tensor

2. scalar 값이 들어왔을 때 차이

```
torch.Tensor(3), torch.Tensor(3).dtype
```

output

```
(tensor([-4.7571e-16,  4.3486e-41, -5.1548e-36]), torch.float32)
```

```
torch.tensor(3), torch.tensor(3).dtype
```

output

```
(tensor(3), torch.int64)
```

→ Tensor 는 단순 scalar 값을 넣게 되면 리스트 안에 n개의 데이터가 랜덤으로 들어가지만, tensor는 단순 스칼라 값도 하나의 데이터로 인식함

tensor vs Tensor

3. requires_grad(자동미분) 차이

```
[ ] torch.tensor([2.,3.], requires_grad=True)
```

```
tensor([2., 3.], requires_grad=True)
```

```
torch.Tensor([2.,3.], requires_grad=True)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-63-325ae3743886> in <cell line: 1>()  
----> 1 torch.Tensor([2.,3.], requires_grad=True)
```

```
TypeError: new() received an invalid combination of arguments - got (list, requires_grad=bool), but expected one of:  
* (*, torch.device device)  
  didn't match because some of the keywords were incorrect: requires_grad  
* (torch.Storage storage)  
* (Tensor other)  
* (tuple of ints size, *, torch.device device)  
* (object data, *, torch.device device)
```

```
[ ] torch.Tensor([2.,3.]).requires_grad_(True)
```

```
tensor([2., 3.], requires_grad=True)
```

➔ 자동미분 할 때, tensor에서는 requires_grad
파라미터가 존재하여 파라미터를 사용하면
되지만,
Tensor는 파라미터가 존재하지 않고 함수로
존재함

tensor vs Tensor

4. 데이터가 없는 경우

```
torch.Tensor()
```

```
tensor([])
```

```
torch.tensor()
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-18-ebc3ceaa76d2> in <cell line: 1>()  
----> 1 torch.tensor()
```

```
TypeError: tensor() missing 1 required positional arguments: "data"
```

[STACK OVERFLOW 검색](#)

➔ Tensor 는 data 가 없어도 만들 수 있지만(빈 tensor 가 생성됨)
tensor 는 data 가 반드시 존재해야 만들 수 있음

인덱스 & 슬라이스

```
temp = torch.FloatTensor([1, 2, 3, 4, 5, 6, 7])
print(temp[0], temp[1], temp[-1])          # 인덱스로 접근
print('-----')
print(temp[2:5], temp[4:-1])               # 슬라이스로 접근
```

output

```
tensor(1.) tensor(2.) tensor(7.)
-----
tensor([3., 4., 5.]) tensor([5., 6.])
```

사칙연산

```
v = torch.tensor([1, 2, 3])  
w = torch.tensor([3, 4, 6])  
  
print(torch.add(v,w))  
print(torch.sub(v,w))  
print(torch.mul(v,w))  
print(torch.div(v,w))
```

output

```
tensor([4, 6, 9])  
tensor([-2, -2, -3])  
tensor([ 3,  8, 18])  
tensor([0.3333, 0.5000, 0.5000])
```

차원 조작

view: 텐서의 모양(shape)을 변경하는데 사용함

```
temp = torch.tensor([
    [1, 2], [3, 4]
])

print(temp)
print(temp.shape)
print("-----")

print(temp.view(4,1))
print(temp.view(4,1).shape)
print("-----")

print(temp.view(-1))
print(temp.view(-1).shape)
print("-----")

print(temp.view(2,-1))
print(temp.view(2,-1).shape)
```

```
tensor([[1, 2],
        [3, 4]])
torch.Size([2, 2])
-----
tensor([[1],
        [2],
        [3],
        [4]])
torch.Size([4, 1])
-----
tensor([1, 2, 3, 4])
torch.Size([4])
-----
tensor([[1, 2],
        [3, 4]])
torch.Size([2, 2])
```

transpose: 두개의 차원만 맞바꾸어 전치 수행
permute: 여러 차원(모든 차원) 을 재배치 할 수 있음

```
temp = torch.tensor([ [1, 2, 3], [4, 5, 6] ])
print(temp.transpose(0,1))
```

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

```
temp = torch.tensor([ [[1,2], [3,4], [5,6]] ])
print(temp.permute(0,2,1))
```

```
tensor([[[1, 3, 5],
        [2, 4, 6]]])
```


squeeze & unsqueeze

`squeeze()`: 크기가 1인 차원을 제거하여 텐서의 차원을 축소

`unsqueeze(dim)`: 지정된 위치에 크기가 1인 새로운 차원을 추가

```
a = torch.tensor([[1, 2, 3]])  
  
print(a.shape)  
print(a.squeeze())  
print(a.squeeze().shape)  
print(a.unsqueeze(2))  
print(a.unsqueeze(2).shape)
```

output

```
torch.Size([1, 3])  
tensor([1, 2, 3])  
torch.Size([3])  
tensor([[[1],  
         [2],  
         [3]]])  
torch.Size([1, 3, 1])
```

torch.cat & torch.stack

torch.cat: 주어진 차원을 따라서 텐서를 연결, 차원 증가 x

torch.stack: 새로운 차원을 추가하여 텐서를 쌓아 올림, 차원 추가

```
a = torch.tensor([[1,2,3]])  
b = torch.tensor([[4,5,6]])  
  
print(torch.cat([a,b], dim = 1))  
print(torch.stack([a,b], dim = 0))
```

output

```
tensor([[1, 2, 3, 4, 5, 6]])  
tensor([[[1, 2, 3]],  
        [[4, 5, 6]]])
```

참고문헌

딥러닝 파이토치 교과서 2장

파이토치 모델 구현

서은서

데이터 준비

단순 파일 불러오기

커스텀 데이터셋 생성

파이토치 제공 데이터셋 사용

데이터 준비

단순 파일 불러오기

```
data = pd.read_csv("파일명")
```

```
x = torch.from_numpy(data['x'].values).unsqueeze(dim=1).float  
y = torch.from_numpy(data['y'].values).unsqueeze(dim=1).float
```

커스텀 데이터셋 생성

파이토치 제공 데이터셋 사용

squeeze 함수 : 차원이 1인 차원을 제거
(dim을 지정하지 않으면 1인 차원을 모두 제거)

Ex)

```
x = torch.rand(1, 1, 20, 128)  
x = x.squeeze() # [1, 1, 20, 128] → [20, 128]
```

```
x2 = torch.rand(1, 1, 20, 128)  
x2 = x2.squeeze(dim=1) # [1, 1, 20, 128] → [1, 20, 128]
```

↔ **unsqueeze 함수** : 1인 차원을 생성
(dim을 무조건 지정해주어야 함)

Ex)

```
x = torch.rand(3, 20, 128)  
x = x.unsqueeze(dim=1) # [3, 20, 128] → [3, 1, 20, 128]
```


데이터 준비

단순 파일 불러오기

커스텀 데이터셋 생성

파이토치 제공 데이터셋 사용

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

```
class CustomDataset(Dataset):
```

```
    def __init__(self, csv_file):      # 필요한 변수를 선언하고, 데이터셋의 전처리를 해 주는 함수
        self.data = pd.read_csv(csv_file)
```

```
    def __len__(self):                # 총 샘플의 수를 가져오는 함수
        return len(self.data)
```

```
    def __getitem__(self, index):      # index번째 데이터를 반환하는 함수
        sample = torch.tensor(self.data.iloc[index,0:3]).int()
        label = torch.tensor(self.data.iloc[index,3]).int() # label값이 4번째 열에 존재하는 경우
        return sample, label
```

```
tensor_dataset = CustomDataset('파일명')
dataset = DataLoader(tensor_dataset, batch_size=4, shuffle=True)
```

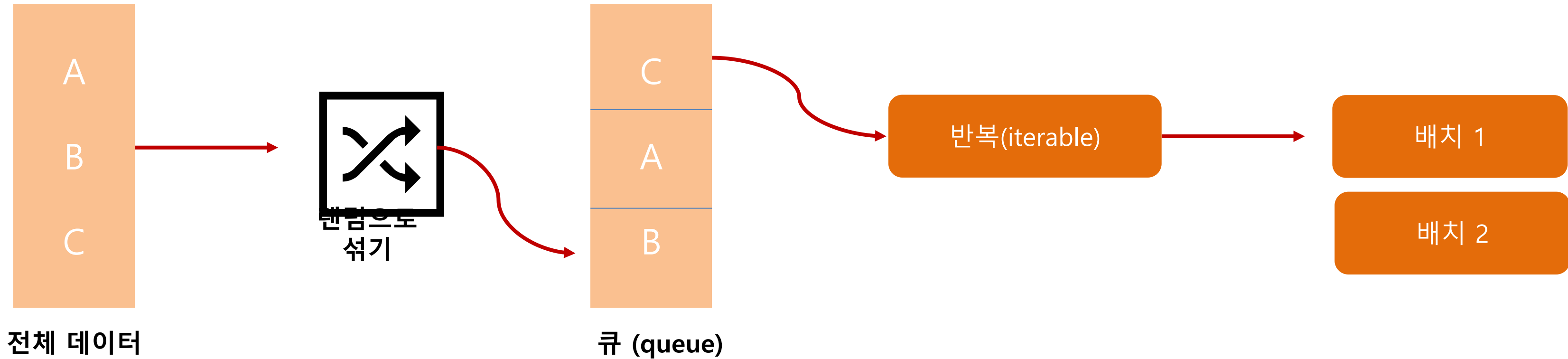
| index | 0 | 1 | 2 | 3 |
|-------|---|---|---|---|
| 1 | | | | |
| ... | | | | |
| 10 | | | | |

sample

label

DataLoader

데이터로더(DataLoader) 객체는 학습에 사용될 데이터 전체를 보관했다가 모델 학습을 할 때 배치 크기만큼 데이터를 꺼내서 사용한다.



```
dataset = DataLoader(tensor_dataset, batch_size=4, shuffle=True)
```

- **batch** : 데이터셋의 전체 데이터가 batch size로 slice되어 공급된다.
- **shuffle** : epoch마다 데이터셋을 섞어, 데이터가 학습되는 순서를 바꾸는 기능 (학습을 할 때는 항상 True로 설정하는 것을 권장)
- **num_worker** : 동시에 처리하는 프로세서의 수 (num_worker 하나를 더 추가하면 20% 정도 속도가 빨라진다.)

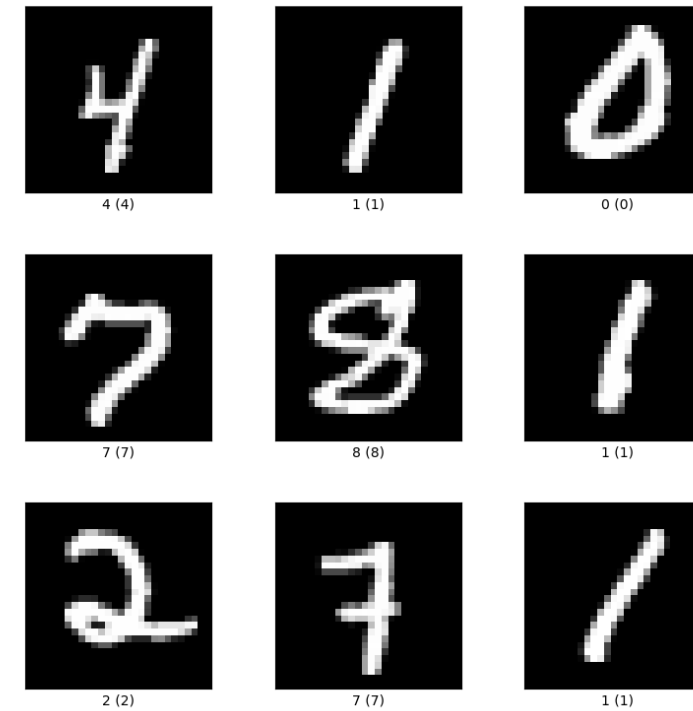
데이터 준비

단순 파일 불러오기

커스텀 데이터셋 생성

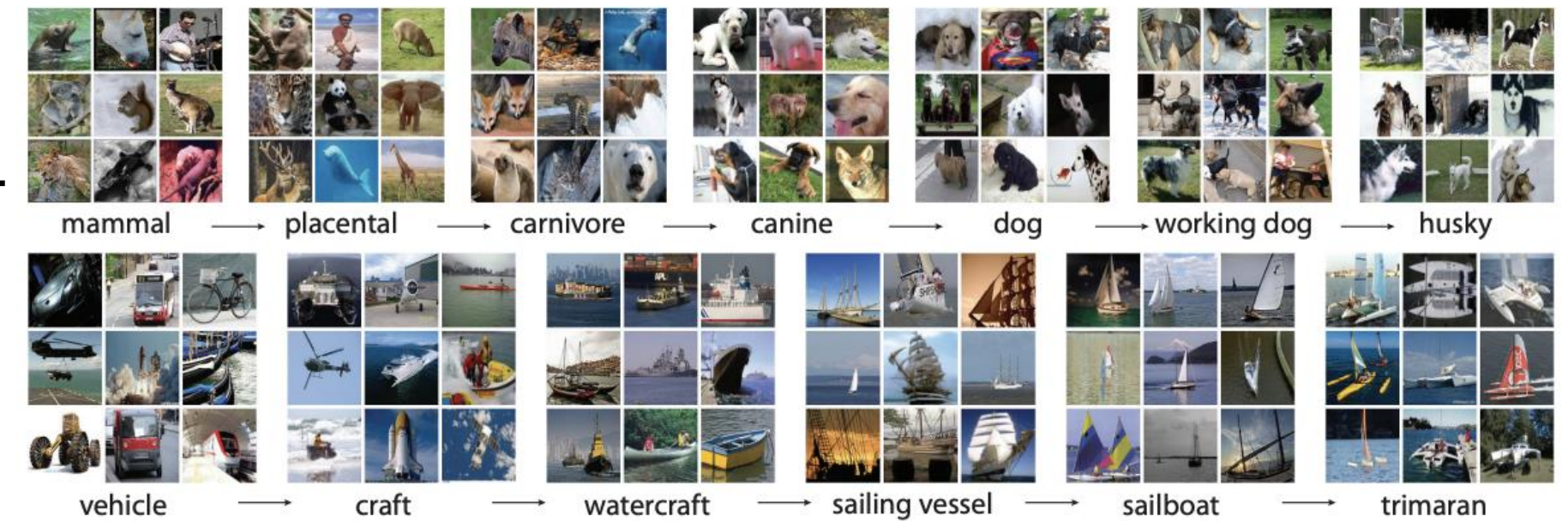
파이토치 제공 데이터셋 사용

MNIST



torchvision

ImageNet



데이터 준비

단순

```
import torchvision.transforms as transforms
from torchvision.datasets import MNIST
import requests
```

```
mnist_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (1.0,))
])
```

Pytorch tensor 타입으로 변환

평균이 0.5, 표준편차가 1.0이 되도록 데이터 분포를 조정

커스텀

```
download_root = '내려받을 경로 지정'
```

```
train_dataset = MNIST(download_root, transform = mnist_transform, train=True, download=True)
```

```
valid_dataset = MNIST(download_root, transform = mnist_transform, train=False, download=True)
```

```
test_dataset = MNIST(download_root, transform = mnist_transform, train=False, download=True)
```

파이토치 제공 데이터셋 사용

모델 정의



① 단순 신경망

② `nn.Module()` 상속

③ `Sequential` 신경망 이용

① 단순 신경망

- nn.Module을 상속받지 않는 매우 단순한 모델

```
model = nn.Linear(in_features = 1, out_features = 1, bias=True)
```

* **nn.Module**: 모든 모델의 근간이 되는 기본 클래스로 forward, backward 등의 함수를 포함하고 있다.

② nn.Module() 상속

- nn.Module을 상속 받기 때문에 기본적으로 `__init__()`과 `forward()` 함수를 사용할 수 있다.

```
import torch.nn as nn
import torch.nn.functional as F
```

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

```
    def __init__(self):
```

```
        super().__init__() # nn.Module을 상속 받는다.
```

```
        self.conv1 = nn.Conv2d(1, 20, 5)
```

```
        self.conv2 = nn.Conv2d(20, 20, 5)
```

}] **__init__** : 모델에서 사용될 모듈, 활성화 함수 등을 정의

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

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

```
        x = F.relu(x)
```

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

```
        x = F.relu(x)
```

```
        return x
```

어떻게 forward propagation을 할지 정의

}] **forward()** : 모델에서 실행되어야 하는 연산을 정의

③ Sequential 신경망 이용

- Sequential 객체 안에 포함된 각 모듈을 순차적으로 실행해줌
- 가독성이 뛰어나게 코드로 작성할 수 있음
- nn.Sequential은 모델의 계층이 복잡할수록 효과가 더 뛰어나다!

```
import torch.nn as nn
```

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

```
    def __init__(self):
```

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

```
        layer= nn.Sequential(  
            nn.Conv2d(1,20,5),  
            nn.ReLU(),  
            nn.Conv2d(20,64,5),  
            nn.ReLU())
```

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

```
        return self.layer(x)
```

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

```
    def __init__(self):
        super(MyNeuralNetwork, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=30, kernel_size=5)
        self.fc1 = nn.Linear(in_features=30*5*5, out_features=128, bias=True)
        self.fc2 = nn.Linear(in_features=128, out_features=10, bias=True)

    def forward(self, x):
        x = F.relu(self.conv1(x), inplace=True)
        x = F.max_pool2d(x, (2, 2))

        x = F.relu(self.conv2(x), inplace=True)
        x = F.max_pool2d(x, (2, 2))

        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x), inplace=True)
        x = F.relu(self.fc2(x), inplace=True)

        return x
```

▲ nn.Sequential을 사용하지 않은 신경망

nn.Sequential을 사용한 신경망 ►

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

```
    def __init__(self):
        super(MyNeuralNetwork, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=30, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        self.layer3 = nn.Sequential(
            nn.Linear(in_features=30*5*5, out_features=128, bias=True),
            nn.ReLU(inplace=True)
        )

        self.layer4 = nn.Sequential(
            nn.Linear(in_features=128, out_features=10, bias=True),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.shape[0], -1)
        x = self.layer3(x)
        x = self.layer4(x)

        return x
```


Quiz

```
class SimpleModel(nn.Module):  
    def __init__(self, input_size, hidden_size, output_size):  
        super(SimpleModel, self).__init__()  
        self.layer1 = nn.Linear(input_size, hidden_size)  
        self.layer2 = nn.Linear(hidden_size, output_size)  
        self.activation = nn.ReLU()  
  
    def forward(self, x):  
  
        x = ①  
        x = ②  
        x = ③  
  
        return x
```

layer1 → 활성화함수 → layer2의 순서로 연산을 진행하고자 할 때,
① ② ③에 들어가야할 코드를 작성해주세요.

모델 훈련

모델, 손실 함수, 옵티마이저
정의



`optimizer.zero_grad()` : 전방향 학습, 기울기 초기화



`output = model(input)` : 출력 계산



`loss = loss_fn(output, target)` : 오차 계산



`loss.backward()` : 역전파 학습



`optimizer.step()` : 기울기 업데이트

DataLoader

모델, 손실 함수, 옵티마이저
정의

```
from torch.optim import optimizer

model = Model()
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer=optimizer, lr_lambda=lambda epoch: 0.95 ** epoch)
```

옵티마이저

데이터와 손실 함수를 바탕으로 모델의 업데이트 방법을
결정

- **step()** : 전달받은 파라미터를 업데이트
- **zero_grad()** : 옵티마이저에 사용된 파라미터들의 기울기를 0으로 만들어 준다.
- **torch.optim.lr_scheduler** : 에포크에 따라 학습률(learning rate)을 조절할 수 있다.

학습률 스케줄러

- 미리 지정한 횟수의 에포크를 지날 때마다 학습률을 감소시킨다.
- 학습률 스케줄러를 이용하면 학습 초기에는 빠른 학습을 진행하다가 '전역 최소점' 근처에 다다르면 학습률을 줄여서 최적점을 찾아갈 수 있도록 해준다.

DataLoader

`optimizer.zero_grad()` : 전방향 학습, 기울기 초기화

파이토치는 기울기 값을 계산하기 위해 `loss.backward()` 메서드를 이용하는데, 이것을 사용하면 새로운 기울기 값이 **이전 기울기 값에 누적되기** 때문에 **초기화**가 필요하다. (단, RNN을 구현할 때는 효과적!)

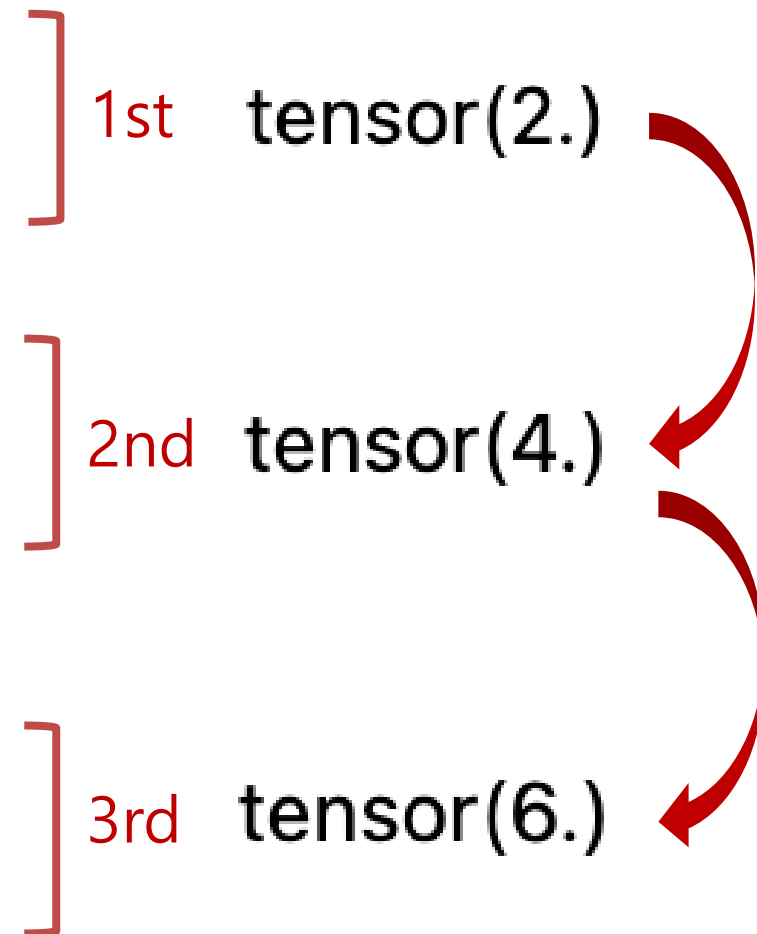
```
import torch
```

```
w = torch.tensor(2.0,requires_grad=True)
```

```
z = 2*w  
z.backward()  
print(w.grad)
```

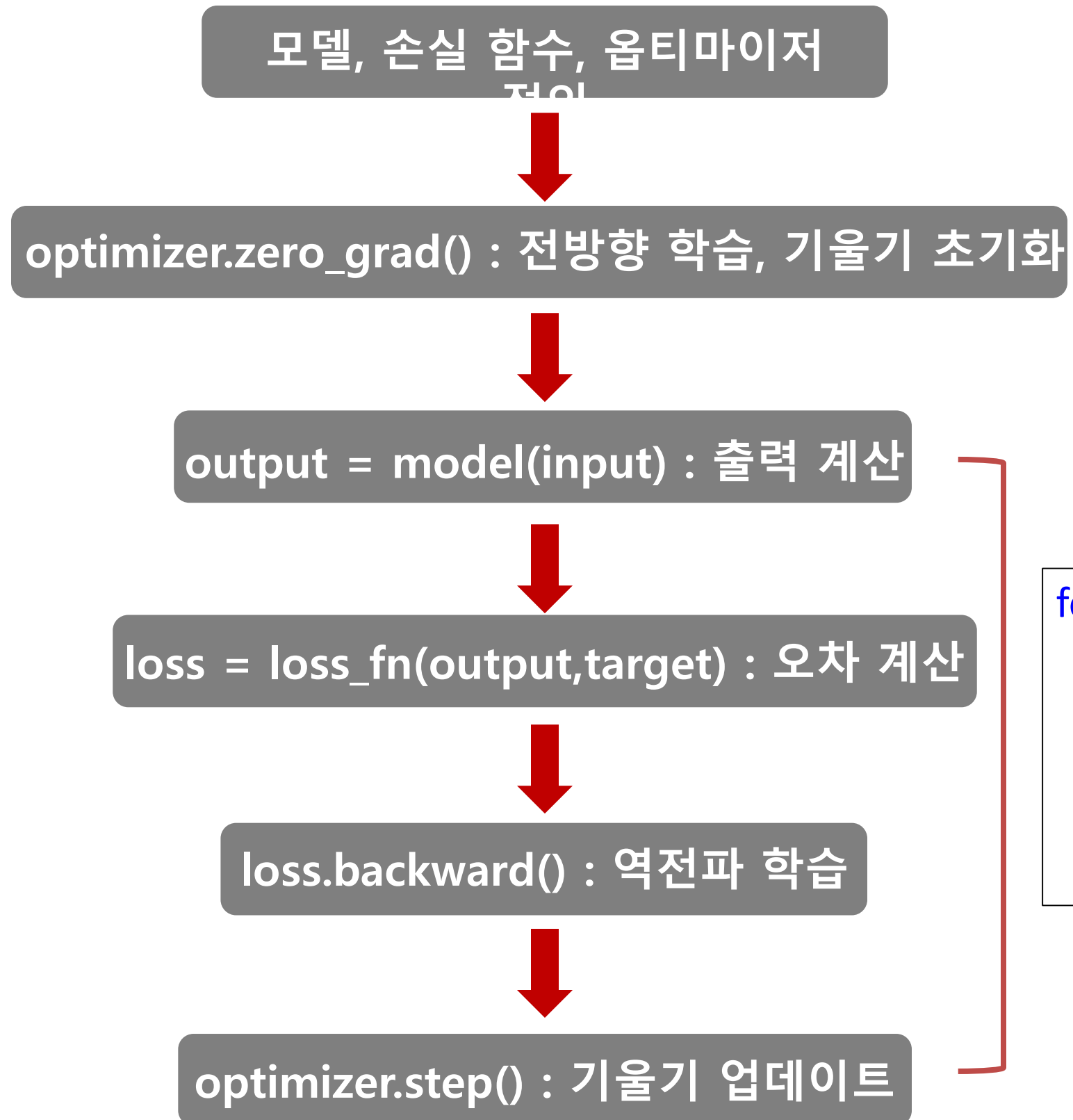
```
z = 2*w  
z.backward()  
print(w.grad)
```

```
z = 2*w  
z.backward()  
print(w.grad)
```



기울기가 누적되고 있음!!!

모델 훈련



```
for epoch in range(100):  
    yhat = model(X_train)  
    loss = criterion(yhat, y_train)  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()
```

파이토치 실습

권도영

데이터

데이터 다운로드: <https://archive.ics.uci.edu/dataset/19/car+evaluation>

컬럼 소개

- 1. price: 자동차 가격
- 2. maint: 자동차 유지 비용
- 3. doors: 자동차 문 개수
- 4. persons: 수용 인원
- 5. lug_capacity: 수하물 용량
- 6. safety: 안전성
- 7. output: 차 상태 [unacc(허용 불가능한 수준), acc(허용 가능한 수준), good(양호), vgood(매우 좋은)]

| price | maint | doors | persons | lug_capacity | safety | output |
|-------|-------|-------|---------|--------------|--------|--------|
| vhigh | vhigh | 2 | 2 | small | low | unacc |
| vhigh | vhigh | 2 | 2 | small | med | unacc |
| vhigh | vhigh | 2 | 2 | small | high | unacc |

1. price: 자동차 가격
2. maint: 자동차 유지 비용
3. doors: 자동차 문 개수
4. persons: 수용 인원
5. lug_capacity: 수하물 용량
6. safety: 안전성



output(차 상태) 칼럼 값을 예측

라이브러리 호출

```
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

라이브러리 설명

- **scikit-learn** : 다양한 머신 러닝 알고리즘을 적용할 수 있는 함수를 제공하는 머신 러닝 라이브러리
- **numpy** : 벡터 및 행렬 연산에서 매우 편리한 기능을 제공하는 파이썬 라이브러리 패키지
- **pandas** : 데이터 처리를 위해 널리 사용되는 파이썬 라이브러리 패키지
- **matplotlib** : 2D, 3D 형태의 플롯(그래프)을 그릴 때 주로 사용하는 패키지(모듈)
- **seaborn** : 데이터 프레임으로 다양한 통계 지표를 표현할 수 있는 시각화 차트를 제공

데이터 호출

```
dataset = pd.read_csv('car_evaluation.csv')  
dataset.head()
```

| | price | maint | doors | persons | lug_capacity | safety | output |
|---|-------|-------|-------|---------|--------------|--------|--------|
| 0 | vhigh | vhigh | 2 | 2 | small | low | unacc |
| 1 | vhigh | vhigh | 2 | 2 | small | med | unacc |
| 2 | vhigh | vhigh | 2 | 2 | small | high | unacc |
| 3 | vhigh | vhigh | 2 | 2 | med | low | unacc |
| 4 | vhigh | vhigh | 2 | 2 | med | med | unacc |

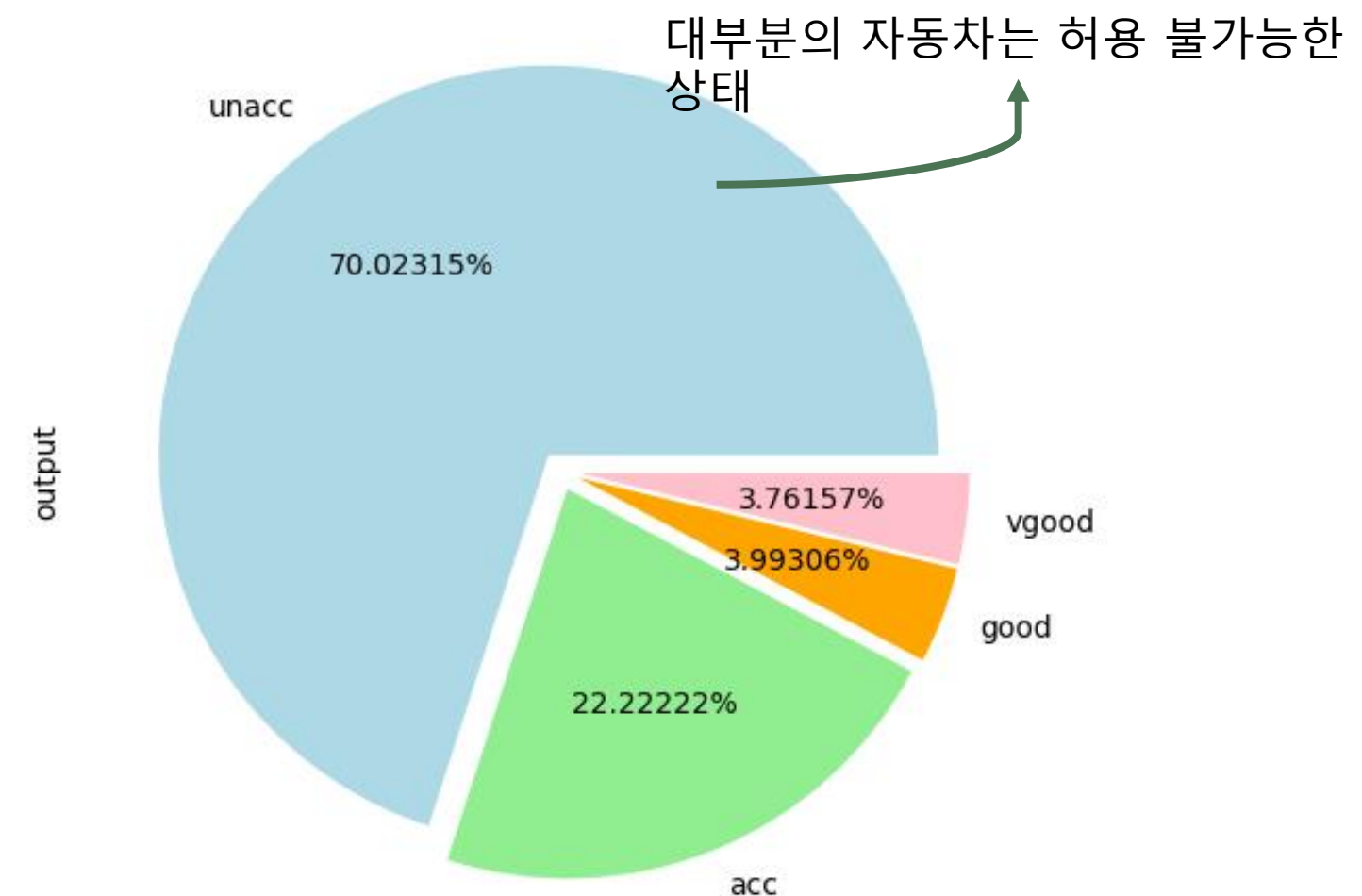


단어를 벡터로 바꾸어 주는 임베딩(embedding) 처리가 필요

데이터 분포 형태 시각화

```
fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 8
fig_size[1] = 6
plt.rcParams["figure.figsize"] = fig_size
dataset.output.value_counts().plot(kind='pie', autopct='%0.05f%%',
                                    colors=['lightblue', 'lightgreen', 'orange', 'pink'],
                                    explode=(0.05, 0.05, 0.05, 0.05))
```

데이터프레임 'dataset'의 'output' 열에 대한 값의 빈도를 계산하고,
그 빈도를 기반으로 차트를 생성



전처리

① 데이터를 범주형(category) 타입으로 변환

```
#예제 데이터셋 칼럼들의 목록
categorical_columns = ['price', 'maint', 'doors', 'persons', 'lug_capacity', 'safety']

#astype() 메서드를 이용하여 데이터를 범주형으로 변환
for category in categorical_columns:
    dataset[category] = dataset[category].astype('category')
```

② 범주형 타입을 텐서로 변환

범주형 데이터를 넘파이 배열로 변환

[범주형 데이터 → dataset[category] → 넘파이 배열 → 텐서]

```
price = dataset['price'].cat.codes.values
maint = dataset['maint'].cat.codes.values
doors = dataset['doors'].cat.codes.values
persons = dataset['persons'].cat.codes.values
lug_capacity = dataset['lug_capacity'].cat.codes.values
safety = dataset['safety'].cat.codes.values
```

범주형 데이터를 숫자(넘파이 배열)로 변환하기 위해 사용

전처리

② 범주형 타입을 텐서로 변환

6개의 넘파이 배열을 np.stack을 사용하여 합침

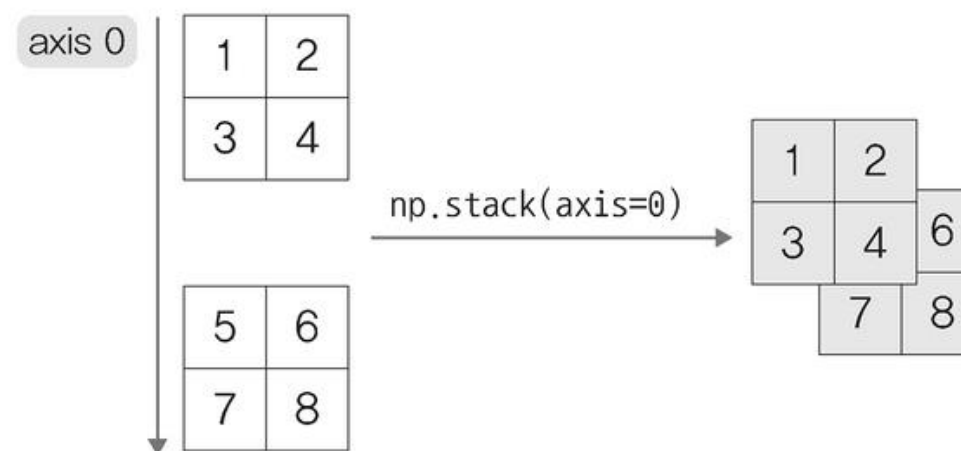
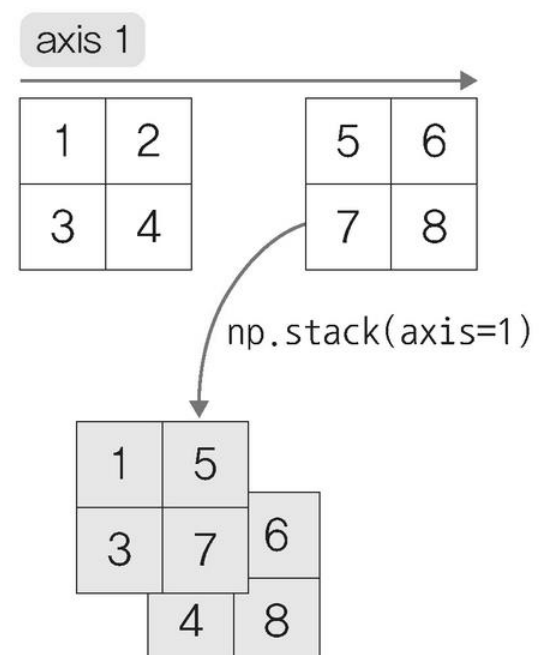
```
categorical_data = np.stack([price, maint, doors, persons, lug_capacity, safety], 1)
```

- **np.stack** : 두 개 이상의 넘파이 객체를 합칠 때 사용

배열들을 새로운 축으로 합쳐 줌

ex) 1차원 배열들을 합쳐서 2차원 배열을 만들거나 2차원 배열 여러 개를 합쳐 3차원 배열을 만들

→ 반드시 두 배열의 차원이 동일해야 함



전처리

② 범주형 타입을 텐서로 변환

6개의 넘파이 배열을 np.stack을 사용하여 합침

```
#합친 넘파이 배열 중 10개의 행을 출력  
categorical_data[:10]
```

[출력 결과]

```
array([[3, 3, 0, 0, 2, 1],  
       [3, 3, 0, 0, 2, 2],  
       [3, 3, 0, 0, 2, 0],  
       [3, 3, 0, 0, 1, 1],  
       [3, 3, 0, 0, 1, 2],  
       [3, 3, 0, 0, 1, 0],  
       [3, 3, 0, 0, 0, 1],  
       [3, 3, 0, 0, 0, 2],  
       [3, 3, 0, 0, 0, 0],  
       [3, 3, 0, 1, 2, 1]], dtype=int8)
```

전처리

② 범주형 타입을 텐서로 변환

torch 모듈을 이용하여 배열을 텐서로 변환

```
categorical_data = torch.tensor(categorical_data, dtype=torch.int64)  
categorical_data[:10]
```

[출력 결과]

```
tensor([[3, 3, 0, 0, 2, 1],  
        [3, 3, 0, 0, 2, 2],  
        [3, 3, 0, 0, 2, 0],  
        [3, 3, 0, 0, 1, 1],  
        [3, 3, 0, 0, 1, 2],  
        [3, 3, 0, 0, 1, 0],  
        [3, 3, 0, 0, 0, 1],  
        [3, 3, 0, 0, 0, 2],  
        [3, 3, 0, 0, 0, 0],  
        [3, 3, 0, 1, 2, 1]])
```

Quiz 4

② 범주형 타입을 텐서로 변환

레이블로 사용할 output 칼럼을 텐서로 변환

```
outputs = pd.get_dummies(dataset.output)
outputs = outputs.values #shape: (a, b)
outputs = torch.tensor(outputs).flatten() #shape: ([c])

print(categorical_data.shape)
print(outputs.shape)
```

```
dataset.shape
```

```
(1728, 7)
```

```
dataset['output'].nunique()
```

```
4
```

Q. a, b, c에 순서대로 들어갈 숫자는 무엇일까요?

범주형 컬럼의 임베딩 크기 정의

```
categorical_column_sizes = [len(dataset[column].cat.categories) for column in categorical_columns]
categorical_embedding_sizes = [(col_size, min(50, (col_size+1)//2)) for col_size in categorical_column_sizes]

# (모든 범주형 칼럼의 고유 값 수, 차원의 크기)
print(categorical_embedding_sizes)
```

↓
주로 칼럼의 고유 값 수를 2로
나눔

[출력 결과]

```
[(4, 2), (4, 2), (4, 2), (3, 2), (3, 2), (3, 2)]
```

데이터셋 분리

데이터셋을 훈련과 테스트 용도로 분리

```
total_records = 1728      #전체 데이터셋의 총 레코드 수
test_records = int(total_records * .2)    #전체 데이터 중 20%를 테스트 용도로 사용

categorical_train_data = categorical_data[:total_records-test_records]
categorical_test_data = categorical_data[total_records-test_records:total_records]
train_outputs = outputs[:total_records-test_records]
test_outputs = outputs[total_records-test_records:total_records]

print(len(categorical_train_data))
print(len(train_outputs))
print(len(categorical_test_data))
print(len(test_outputs))
```

[출력 결과]

```
1383
1383
345
345
```

모델의 네트워크 생성

```
class Model(nn.Module): # 클래스로 구현되는 모델은 nn.Module을 상속받음
    def __init__(self, embedding_size, output_size, layers, p=0.4):
        super().__init__()
        self.all_embeddings = nn.ModuleList([nn.Embedding(ni, nf) for ni, nf in embedding_size])
        self.embedding_dropout = nn.Dropout(p)

        all_layers = []
        num_categorical_cols = sum((nf for ni, nf in embedding_size))
        input_size = num_categorical_cols

        for i in layers:
            all_layers.append(nn.Linear(input_size, i))
            all_layers.append(nn.ReLU(inplace=True))
            all_layers.append(nn.BatchNorm1d(i))
            all_layers.append(nn.Dropout(p))
            input_size = i

        all_layers.append(nn.Linear(layers[-1], output_size))
        self.layers = nn.Sequential(*all_layers)
```

모델의 네트워크 생성

`__init__()`

: 모델에서 사용될 파라미터와 신경망을 초기화하기 위한 용도로 사용하며, 객체가 생성될 때 자동으로 호출됨

```
def __init__(self, embedding_size, output_size, layers, p=0.4)
```

(a) (b) (c) (d) (e)

- ① **self** : 인스턴스 자기 자신을 의미
- ② **embedding_size** : 범주형 칼럼의 임베딩 크기
- ③ **output_size** : 출력층의 크기
- ④ **layers** : 모든 계층에 대한 목록
- ⑤ **P**: 드롭아웃(기본값은 0.5)

*드롭아웃: 뉴런을 임의로 삭제하면서 학습하는 방법

모델의 네트워크 생성

```
class Model(nn.Module): # 클래스로 구현되는 모델은 nn.Module을 상속받음
    def __init__(self, embedding_size, output_size, layers, p=0.4):
        super().__init__() # 부모 클래스에 접근할 때 사용. Self 사용 X
        self.all_embeddings = nn.ModuleList([nn.Embedding(ni, nf) for ni, nf in embedding_size])
        self.embedding_dropout = nn.Dropout(p)

        all_layers = []
        num_categorical_cols = sum((nf for ni, nf in embedding_size))
        input_size = num_categorical_cols # 입력층의 크기를 찾기 위해 범주형 칼럼 개수를 input_size 변수에 저장

        for i in layers: # 각 계층을 all_layers 목록에 추가
            all_layers.append(nn.Linear(input_size, i))
            all_layers.append(nn.ReLU(inplace=True))
            all_layers.append(nn.BatchNorm1d(i))
            all_layers.append(nn.Dropout(p))
            input_size = i

        all_layers.append(nn.Linear(layers[-1], output_size))
        self.layers = nn.Sequential(*all_layers) # 신경망의 모든 계층이 순차적으로 실행되도록 all_layers를 nn.Sequential 클래스로
```

전달

- **Linear** : 입력 데이터에 선형 변환을 진행한 결과 (입력과 가중치를 곱한 후 바이어스를 더한 결과)
- **ReLu** : 활성화 함수로 사용
- **BatchNorm1d** : 배치 정규화 용도로 사용
(신경망 안에서 데이터의 평균과 분산을 조정하는것. 일반적으로 평균이 0, 분산이 1이 되도록 정규화함)
- **Dropout** : 과적합 방지에 사용

모델의 네트워크 생성

```
class Model(nn.Module):  
    def forward(self, x_categorical):  
        embeddings = []  
        for i,e in enumerate(self.all_embeddings):  
            embeddings.append(e(x_categorical[:,i]))  
        x = torch.cat(embeddings, 1)  
        x = self.embedding_dropout(x)  
        x = self.layers(x)  
        return x
```

forward() 함수 : 학습 데이터를 입력받아 연산을 진행
모델 객체를 데이터와 함께 호출하면 자동으로 실행됨

Model 클래스의 객체 생성

```
model = Model(categorical_embedding_sizes, 4, [200, 100, 50], p=0.4)
print(model)
```

[출력 결과]

```
Model(
  (all_embeddings): ModuleList(
    (0): Embedding(4, 2)
    (1): Embedding(4, 2)
    (2): Embedding(4, 2)
    (3): Embedding(3, 2)
    (4): Embedding(3, 2)
    (5): Embedding(3, 2)
  )
  (embedding_dropout): Dropout(p=0.4, inplace=False)
  (layers): Sequential(
    (0): Linear(in_features=12, out_features=200, bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): Dropout(p=0.4, inplace=False)
    (8): Linear(in_features=100, out_features=50, bias=True)
    (9): ReLU(inplace=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): Dropout(p=0.4, inplace=False)
    (12): Linear(in_features=50, out_features=4, bias=True)
  )
)
```

모델의 파라미터 정의

```
loss_function = nn.CrossEntropyLoss() # 크로스 엔트로피 손실 함수 사용
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) #옵티마이저로 아담(Adam) 사용
```

- **Cross Entropy** : 모델의 예측이 실제 레이블과 얼마나 일치하는지를 측정
주로 이진분류와 다중 클래스 분류에서 사용
모델이 높은 확률로 올바른 클래스를 예측할 때 손실이 감소하며, 모델이 틀린 예측을 할 때 손실이 증가
- **Adam** : 딥러닝 최적화 기법
기울기 최적화 방법과 모멘텀 방법을 결합하여 최적화 성능을 향상시킴

CPU/GPU 사용 지정

```
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

GPU가 있다면 GPU를 사용하고, 없다면 CPU를 사용하도록 설정

모델 학습

```
epochs = 500 # 전체 데이터셋을 몇 번 반복하여 모델을 훈련시킬 것인지 결정
aggregated_losses = [] # 각 에포크에서의 손실을 저장하는 리스트

# 훈련 데이터의 레이블을 모델의 디바이스로 이동시킴
train_outputs = train_outputs.to(device=device, dtype=torch.int64)
for i in range(epochs): # for문은 500회 반복됨. 각 반복마다 손실 함수가 오차를 계산
    i += 1
    y_pred = model(categorical_train_data).to(device) # 현재 모델을 사용하여 훈련 데이터의 예측값을 계산
    single_loss = loss_function(y_pred, train_outputs) # 예측값과 실제 레이블 간의 손실을 계산
    aggregated_losses.append(single_loss) # 반복할 때마다 오차를 aggregated_losses에 추가

    if i%25 == 1:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

optimizer.zero_grad() # 옵티마이저의 기울기를 초기화
single_loss.backward() # 가중치를 업데이트하기 위해 손실 함수의 backward() 메서드 호출
optimizer.step() # 옵티마이저 함수의 step() 메서드를 이용하여 기울기 업데이트

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}') # 오차가 25 에포크마다 출력됨
```

모델 학습

[출력 결과]

```
epoch: 1 loss: 1.58690417
epoch: 26 loss: 1.32208347
epoch: 51 loss: 1.25037670
epoch: 76 loss: 1.14533412
epoch: 101 loss: 1.04376388
epoch: 126 loss: 0.94333309
epoch: 151 loss: 0.82459933
epoch: 176 loss: 0.75102794
epoch: 201 loss: 0.70688218
epoch: 226 loss: 0.67204970
epoch: 251 loss: 0.65042794
epoch: 276 loss: 0.62593251
epoch: 301 loss: 0.61263412
epoch: 326 loss: 0.60477704
epoch: 351 loss: 0.58988392
epoch: 376 loss: 0.58695173
epoch: 401 loss: 0.57759738
epoch: 426 loss: 0.57076532
epoch: 451 loss: 0.57889175
epoch: 476 loss: 0.56059849
epoch: 500 loss: 0.5716277361
```



25 에포크마다 출력된 오차 정보를 보여줌

테스트 데이터셋으로 모델 예측

```
# 테스트 데이터의 레이블을 모델의 디바이스(device)로 이동시킴
test_outputs = test_outputs.to(device=device, dtype=torch.int64)
# 연산 추적을 비활성화함
with torch.no_grad():
    # 테스트 데이터를 모델에 전달하여 예측값을 계산
    y_val = model(categorical_test_data).to(device) # 예측값은 y_val에 저장됨
    loss = loss_function(y_val, test_outputs) # 계산된 예측값과 실제 테스트 레이블 test_outputs 간의 손실을 계산
print(f'Loss: {loss:.8f}')
```

[출력 결과]

Loss: 0.55816710

모델 평가

```
import warnings
warnings.filterwarnings('ignore')
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

test_outputs=test_outputs.cpu().numpy()
print(confusion_matrix(test_outputs,y_val))
print(classification_report(test_outputs,y_val))
print(accuracy_score(test_outputs, y_val))
```

[출력 결과]

```
[[258  1]
 [ 86  0]]
```

| | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0 | 0.75 | 1.00 | 0.86 | 259 |
| 1 | 0.00 | 0.00 | 0.00 | 86 |
| accuracy | | | 0.75 | 345 |
| macro avg | 0.38 | 0.50 | 0.43 | 345 |
| weighted avg | 0.56 | 0.75 | 0.64 | 345 |

0.7478260869565218

- **accuracy (정확도)**: 전체 예측 건수에서 정답을 맞힌 건수의 비율
- **recall (재현율)**: 실제로 정답이 1이라고 할 때 모델도 1로 예측한 비율
- **precision (정밀도)**: 모델이 1이라고 예측한 것 중에서 실제로 정답이 1인 비율
- **F1 score**: 정밀도와 재현율의 조화평균

참고문헌

딤러닝 파이토치 교과서 p.69~97