2장
실습 환경 설정과 파이토치 기초

2장 실습 환경 설정과 파이토치 기초

2.3 실습 환경 설정

2.3 실습 환경 설정

아나콘다 설치

1. 다음 웹 사이트에서 아나콘다(Anaconda)를 내려받음

   Download를 누른 후 자신에게 맞는 버전을 내려받음

   책에서는 윈도를 기준으로 설명하므로 64-Bit Graphical Installer를 내려받았음

   macOS에서도 동일하게 진행하면 됨

   https://www.anaconda.com/download

2.3 실습 환경 설정

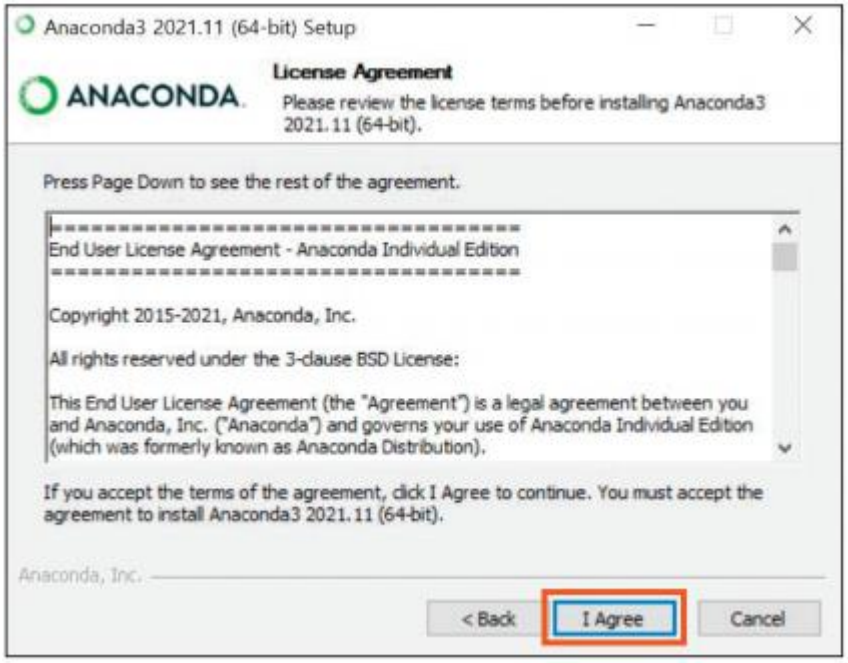
▼ 그림 2-16 설치 시작

2.3 실습 환경 설정

- **아나콘다 설치**

  3. 라이선스 동의 화면이 나오면 **I Agree**를 누름

▼ **그림 2-17 라이선스 동의**

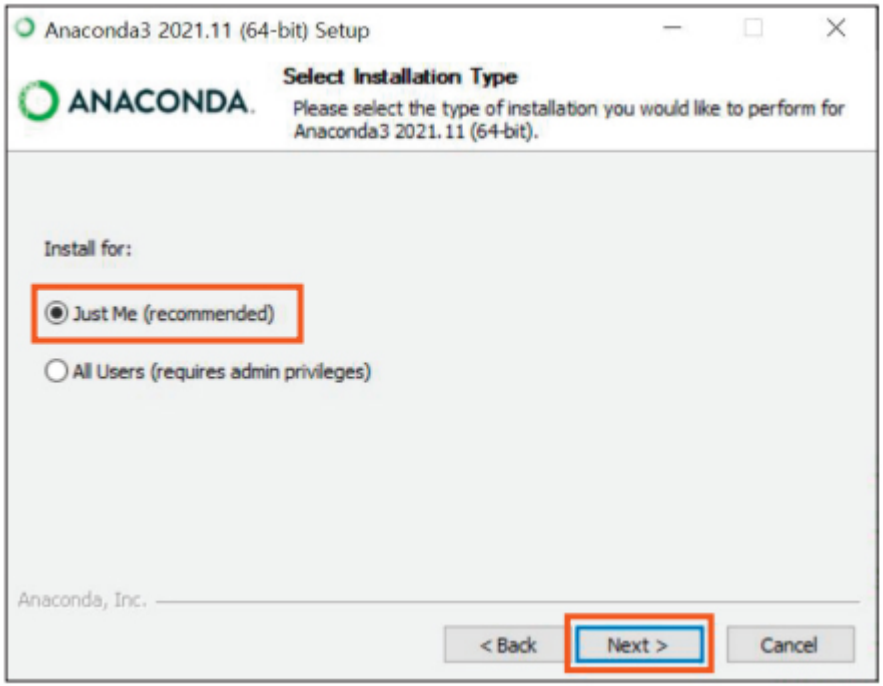


2.3 실습 환경 설정

아나콘다 설치

4. 다음 화면이 나오면 **Just Me**를 선택하고 **Next**를 누름

▼ **그림 2-18 설치 유형 선택**

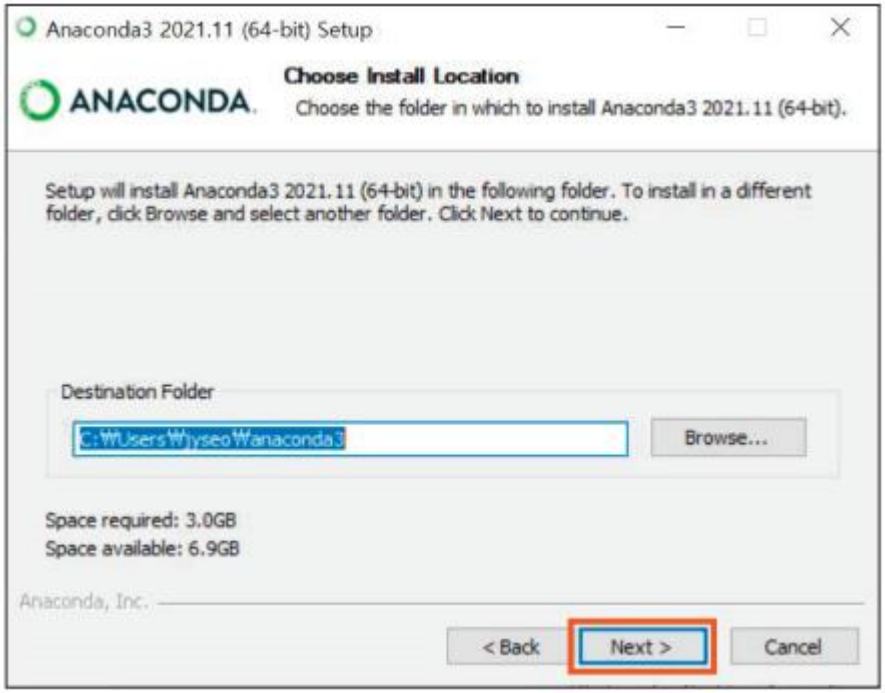


2.3 실습 환경 설정

● 아나콘다 설치

5. 설치 경로를 선택하는 화면이 나오면 기본값으로 두고 **Next**를 누름
   원하는 경로로 변경해도 됨

▼ **그림 2-19 설치 경로 선택**

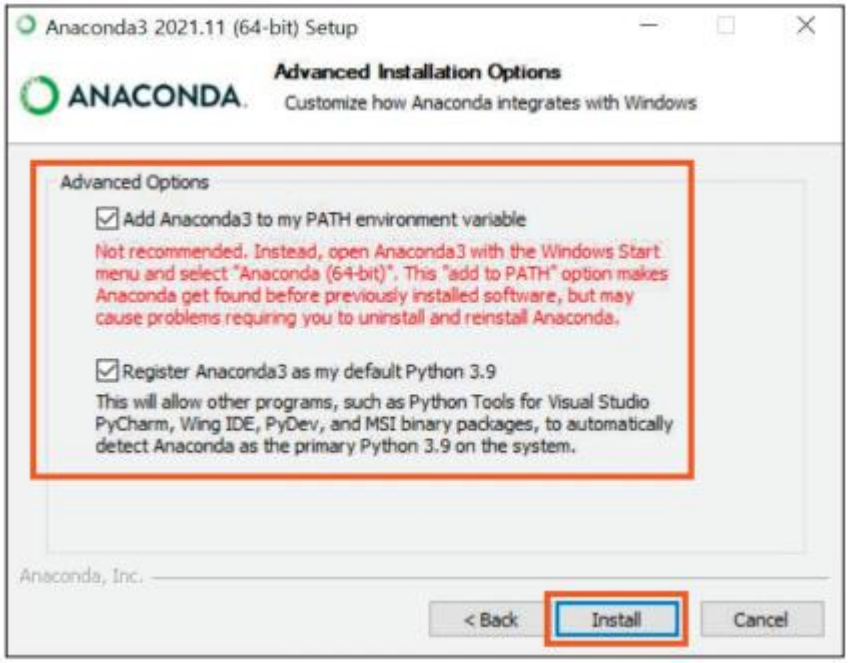


2.3 실습 환경 설정

- **아나콘다 설치**

  6. 다음 화면이 나오면 옵션 두 개를 모두 체크한 후 **Install**을 누름
  첫 번째 옵션을 선택하면 아나콘다 환경 변수가 자동으로 등록

▼ **그림 2-20 설치 시작**

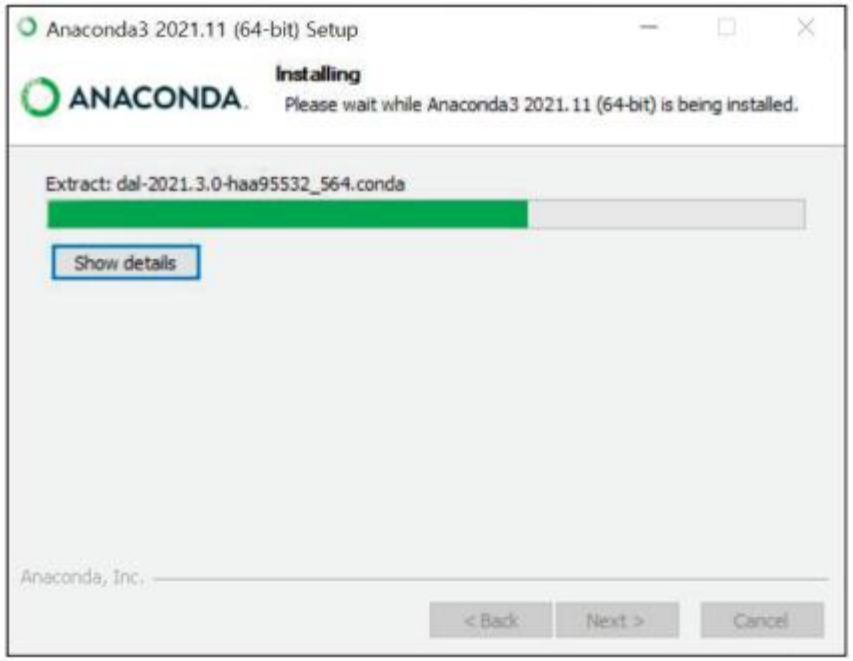


2.3 실습 환경 설정

- **아나콘다 설치**

  7. 다음과 같이 설치가 시작

▼ **그림 2-21 설치 중**



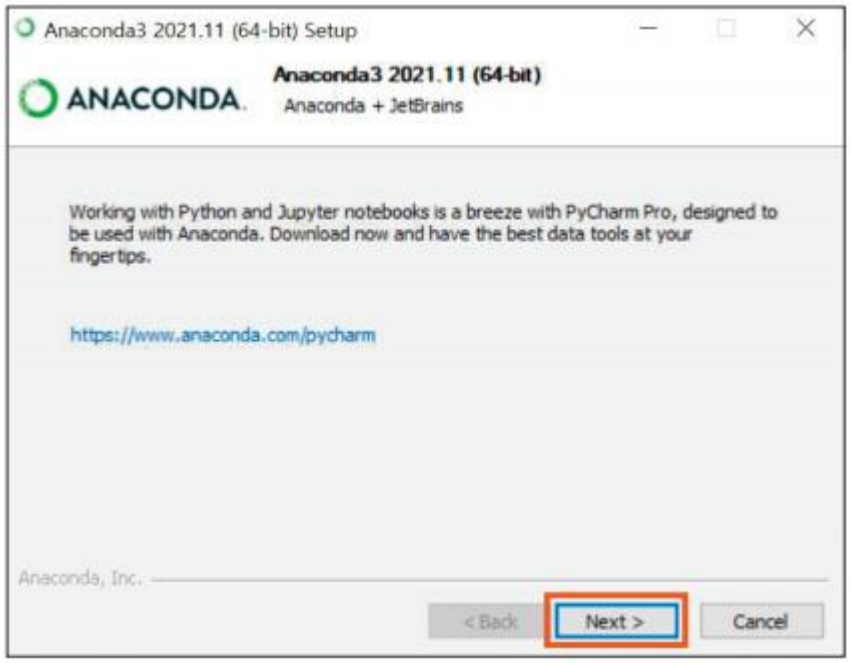
2.3 실습 환경 설정

- **아나콘다 설치**

  8. 설치를 확인한 후 **Next**를 누름

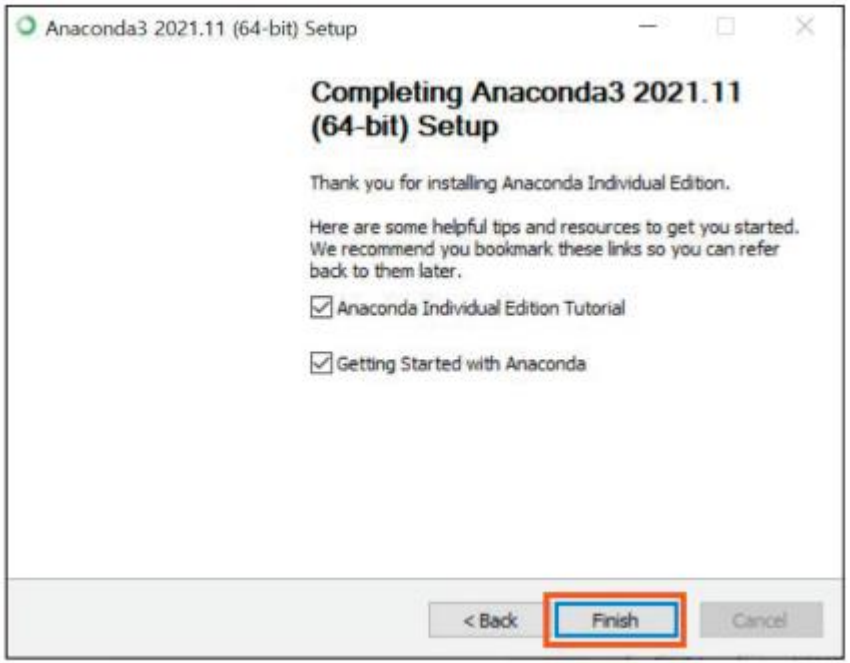
  완료 화면이 나오면 **Finish**로 설치를 완료

▼ **그림 2-22 설치 확인 및 PyCharm 안내**



2.3 실습 환경 설정

▼ 그림 2-23 설치 완료



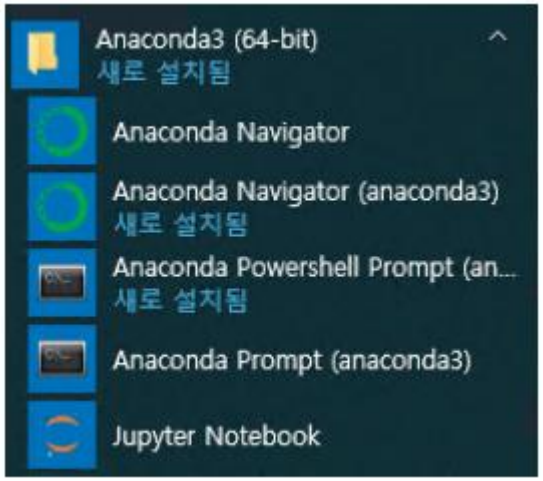
2.3 실습 환경 설정

● 가상 환경 생성 및 파이토치 설치

가상 환경 생성하기

1. 윈도 메뉴 시작 화면에서 **Anaconda3** > **Anaconda Prompt**를 선택

▼ **그림 2-25 아나콘다 프롬프트**



2.3 실습 환경 설정

- **가상 환경 생성 및 파이토치 설치**

2. 가상 환경을 만들어 보자

conda create -n 환경이름 python=3.9.0(설치된 파이썬 버전에 따라 다름) 명령을 이용하여 가상 환경을 생성할 수 있음

다음과 같이 입력하여 'torch_book'이라는 이름의 가상 환경을 만들어 주자

중간에 설치 여부를 묻는다면 'y'를 입력

```
> conda create -n torch_book python=3.9.0
```

파이썬 3.10 이상을 설치하면 파이토치와 호환성 문제가 있기 때문에 3.9 버전을 설치

2.3 실습 환경 설정

- **가상 환경 생성 및 파이토치 설치**

  3. 생성된 가상 환경을 확인

  다음 명령으로 아나콘다의 가상 환경 목록을 확인할 수 있음

```
> conda env list
# conda environments:
#
base e:\Anaconda3
torch_book e:\Anaconda3\envs\torch_book
```

  torch_book 가상 환경이 만들어졌음

2.3 실습 환경 설정

- **가상 환경 생성 및 파이토치 설치**

  4. 다음 명령을 입력하여 가상 환경을 활성화

  ```
  > activate torch_book
  ```

  가상 환경을 잘못 만들어서 삭제하고 싶을 때는 다음 명령으로 삭제할 수 있음

  ```
  > conda env remove -n torch_book
  ```

2.3 실습 환경 설정

- **가상 환경 생성 및 파이토치 설치**

  파이토치 설치하기

  - 아나콘다 프롬프트에서 다음 명령들을 입력하여 파이토치를 설치할 수 있음(다음과 같이 특정 버전을 지정하여 설치해도 무방함)
  - 책에서는 현재 시점의 최신 버전인 1.9.0 버전을 설치

2.3 실습 환경 설정

- **가상 환경 생성 및 파이토치 설치**
  - Pytorch 설치
    - conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
  - 주피터 노트북을 설치
    - pip install jupyter notebook
  - 가상 환경에서 jupyter notebook을 입력하여 주피터 노트북을 실행
    - jupyter notebook

2.1 파이토치 개요

2.1 파이토치 개요

파이토치 개요

파이토치(PyTorch)는 2017년 초에 공개된 딥러닝 프레임워크로 루아(Lua) 언어로 개발되었던 토치(Torch)를 페이스북에서 파이썬 버전으로 내놓은 것

토치는 파이썬의 넘파이(NumPy) 라이브러리처럼 과학 연산을 위한 라이브러리로 공개되었지만 이후 발전을 거듭하면서 딥러닝 프레임워크로 발전

파이토치 공식 튜토리얼에서는 파이토치를 다음과 같이 언급하고 있음

파이썬 기반의 과학 연산 패키지로 다음 두 집단을 대상으로 함

-
- 넘파이를 대체하면서 GPU를 이용한 연산이 필요한 경우
  최대한의 유연성과 속도를 제공하는 딥러닝 연구 플랫폼이 필요한 경우

무엇보다 주목받는 이유 중 하나는 간결하고 빠른 구현성에 있음

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 파이토치 특징은 다음과 같이 한마디로 특징 지을 수 있음

    GPU에서 텐서 조작 및 동적 신경망 구축이 가능한 프레임워크

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - GPU, 텐서, 동적 신경망이란 무엇을 의미할까?
  - 각각의 의미는 다음과 같음

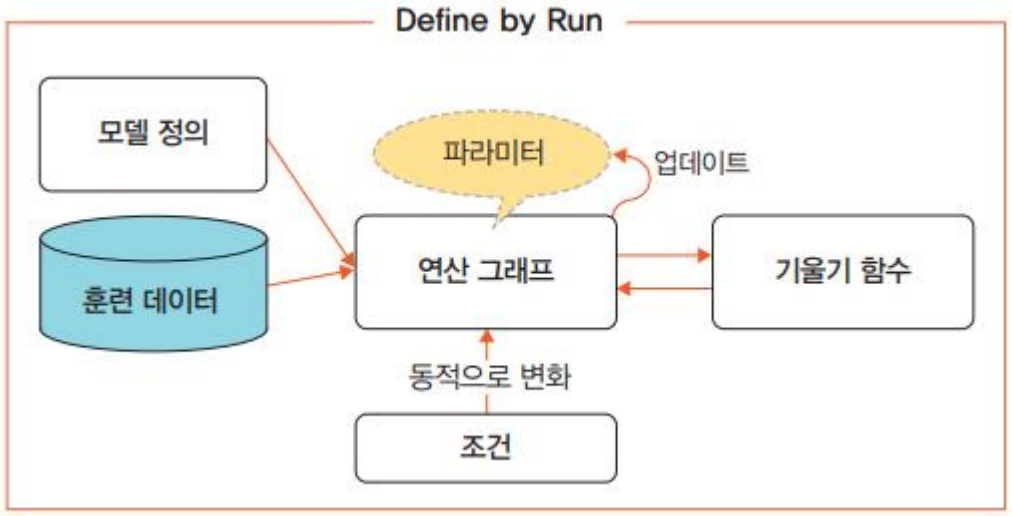
    GPU(Graphics Processing Unit): 연산 속도를 빠르게 하는 역할
    - 딥러닝에서는 기울기를 계산할 때 미분을 쓰는데, GPU를 사용하면 빠른 계산이 가능
    - 내부적으로 CUDA, cuDNN이라는 API를 통해 GPU를 연산에 사용
    - 병렬 연산에서 GPU의 속도는 CPU의 속도보다 훨씬 빠르므로 딥러닝 학습에서 GPU 사용은 필수라고 할 수 있음

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - **텐서**(Tensor): 파이토치에서 텐서 의미는 다음과 같음
    - 텐서는 파이토치의 데이터 형태
    - 텐서는 단일 데이터 형식으로 된 자료들의 다차원 행렬
    - 텐서는 간단한 명령어(변수 뒤에 .cuda()를 추가)를 사용해서 GPU로 연산을 수행하게 할 수 있음
  - **동적 신경망**: 훈련을 반복할 때마다 네트워크 변경이 가능한 신경망을 의미
    - 예를 들어 학습 중에 은닉층을 추가하거나 제거하는 등 모델의 네트워크 조작이 가능
    - 연산 그래프를 정의하는 것과 동시에 값도 초기화되는 'Define by Run' 방식을 사용
    - 연산 그래프와 연산을 분리해서 생각할 필요가 없기 때문에 코드를 이해하기 쉬움

2.1 파이토치 개요

▼ 그림 2-1 파이토치 'Define by Run'



2.1 파이토치 개요

파이토치 특징 및 장점

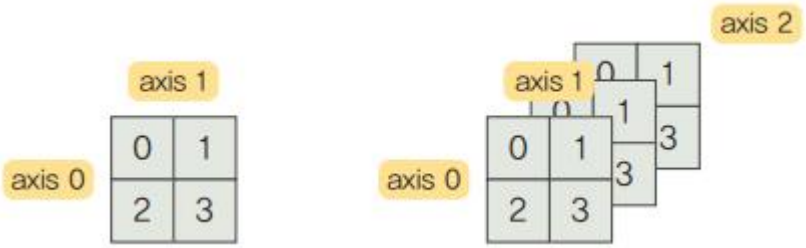
벡터, 행렬, 텐서

- 인공지능(머신 러닝/딥러닝)에서 데이터는 벡터(vector)로 표현
- 벡터는 [1.0, 1.1, 1.2]처럼 숫자들의 리스트로, 1차원 배열 형태
- 행렬(matrix)은 행과 열로 표현되는 2차원 배열 형태
- 이때 가로줄을 행(row)이라고 하며, 세로줄을 열(column)이라고 함
- 마지막으로 텐서는 3차원 이상의 배열 형태
- 이를 정리하면 다음과 같음

  - 1차원 축(행)=axis 0=벡터
  - 2차원 축(열)=axis 1=행렬
  - 3차원 축(채널)=axis 2=텐서

2.1 파이토치 개요

▼ 그림 2-2 벡터, 행렬, 텐서



2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 행렬은 복수의 차원을 가지는 데이터 레코드의 집합
  - 이때 하나의 데이터 레코드를 벡터 단독으로 나타낼 때는 다음과 같이 하나의 열로 표기

$$x_1 = \begin{bmatrix} 1.1 \\ 2.7 \\ 3.3 \\ 0.2 \end{bmatrix} \quad x_2 = \begin{bmatrix} 4.5 \\ 1.2 \\ 0.7 \\ 3.5 \end{bmatrix}$$

  - 반면에 복수의 데이터 레코드 집합을 행렬로 나타낼 때는 다음과 같이 하나의 데이터 레코드가 하나의 행으로 표기

$$X = \begin{bmatrix} 1.1 & 2.7 & 3.3 & 0.2 \\ 4.5 & 1.2 & 0.7 & 3.5 \end{bmatrix}$$

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 즉, 행렬의 일반적인 표현은 다음과 같음

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

  - 텐서는 행렬의 다차원 표현이라고 생각하면 쉬움
  - 같은 크기의 행렬이 여러 개 묶여 있는 것으로 다음과 같이 표현할 수 있음

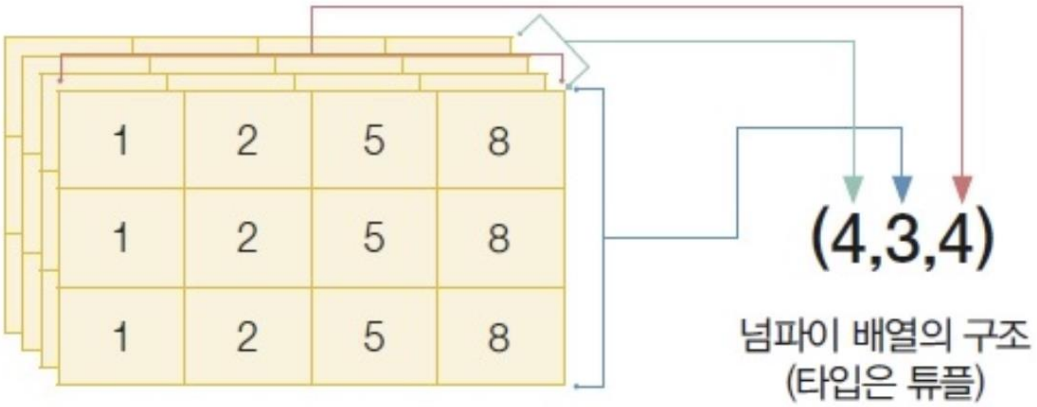


그림 3-6 랭크가 3일 때, 3차원 배열의 구조

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 파이토치에서 텐서를 표현하기 위해서는 다음 코드와 같이 torch.tensor()를 사용

```
import torch
torch.tensor([[1., -1.], [1., -1.]])
```

  - 생성된 텐서의 형태는 다음과 같이 표현

```
tensor([[ 1., -1.],
        [ 1., -1.]])
```

  - 벡터, 행렬 등 자세한 내용은 선형대수학 도서를 참고

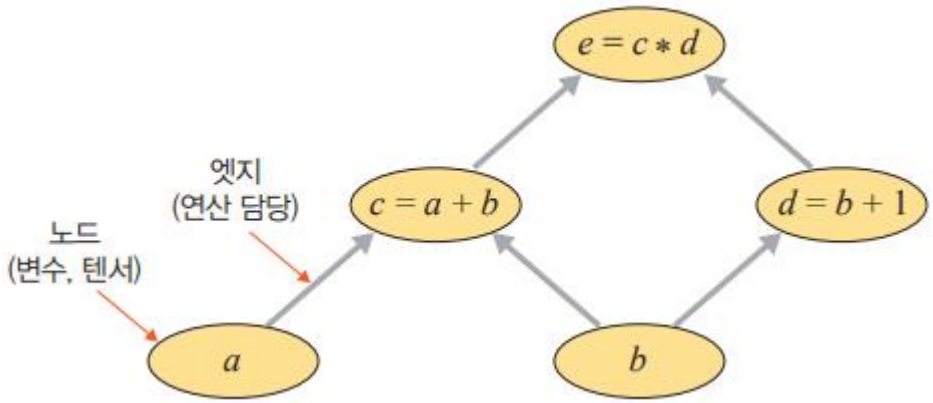
2.1 파이토치 개요

● 파이토치 특징 및 장점

연산 그래프

- 연산 그래프는 방향성이 있으며 변수(예 텐서)를 의미하는 노드와 연산(예 곱하기, 더하기)을 담당하는 엣지로 구성
- 다음 그림과 같이 노드는 변수(a, b)를 가지고 있으며 각 계산을 통해 새로운 텐서(c, d, e)를 구성할 수 있음

▼ **그림 2-3 파이토치 연산 그래프**



2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 신경망은 연산 그래프를 이용하여 계산을 수행
  - 즉, 네트워크가 학습될 때 손실 함수의 기울기가 가중치와 바이어스를 기반으로 계산되며, 이후 경사 하강법을 사용하여 가중치가 업데이트
  - 이때 연산 그래프를 이용하여 이 과정이 효과적으로 수행

2.1 파이토치 개요

- **파이토치 특징 및 장점**
  - 파이토치는 효율적인 계산, 낮은 CPU 활용, 직관적인 인터페이스와 낮은 진입 장벽 등을 장점으로 꼽을 수 있음

    - 단순함(효율적인 계산)
      - 파이썬 환경과 쉽게 통합할 수 있음
      - 디버깅이 직관적이고 간결함

2.1 파이토치 개요

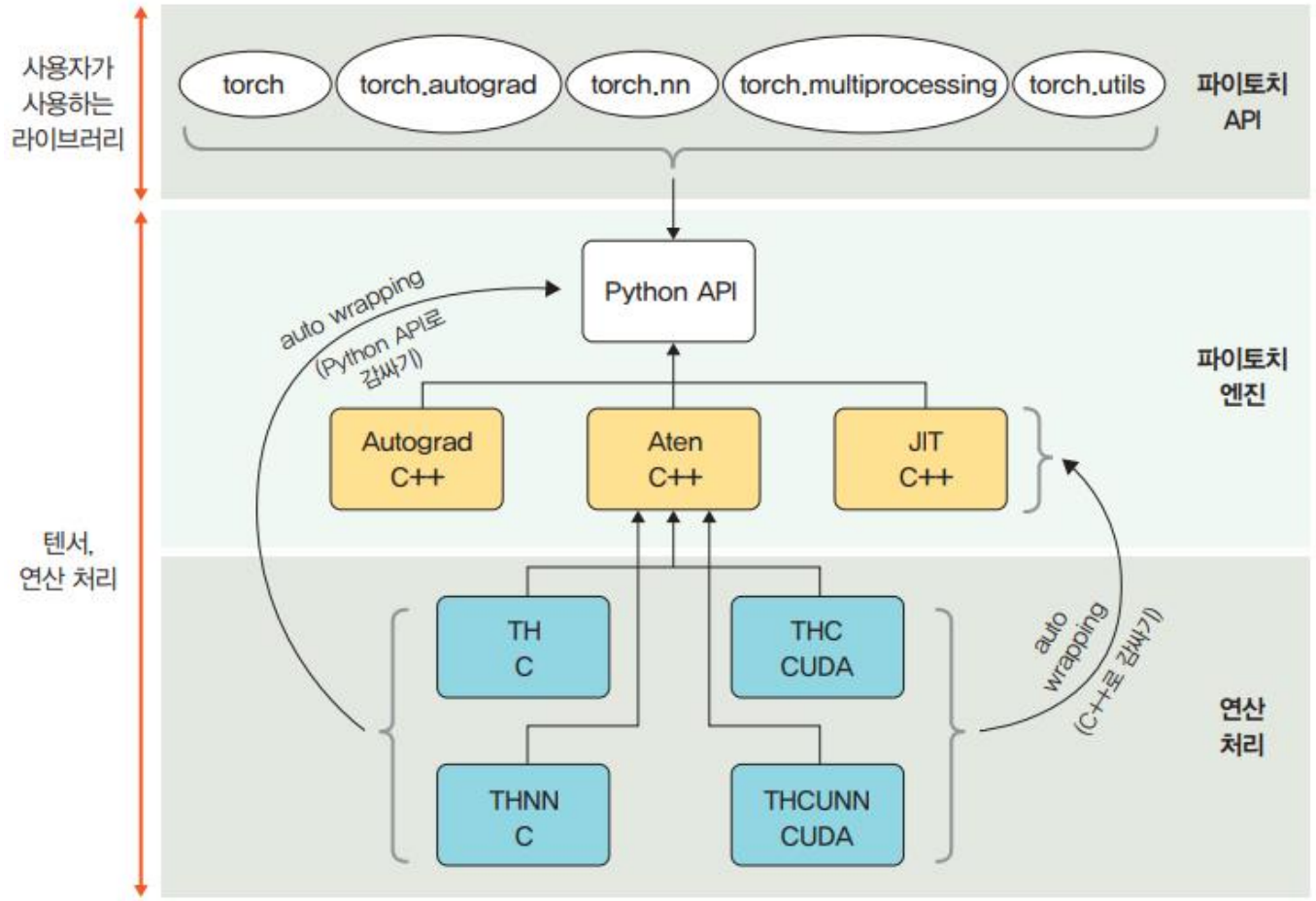
- **파이토치 특징 및 장점**
  - 성능(낮은 CPU 활용)
    - 모델 훈련을 위한 CPU 사용률이 텐서플로와 비교하여 낮음
    - 학습 및 추론 속도가 빠르고 다루기 쉬움
  - 직관적인 인터페이스
    - 텐서플로처럼 잦은 API 변경(예 layers → slim → estimators → tf.keras)이 없어 배우기 쉬움

2.1 파이토치 개요

- **파이토치의 아키텍처**
  - 파이토치의 아키텍처는 간단함
  - 크게 세 개의 계층으로 나누어 설명할 수 있음
  - 가장 상위 계층은 파이토치 API가 위치해 있으며 그 아래에는 파이토치 엔진이 있음
  - 파이토치 엔진에서는 다차원 텐서 및 자동 미분을 처리
  - 마지막으로 가장 아래에는 텐서에 대한 연산을 처리
  - CPU/GPU를 이용하는 텐서의 실질적인 계산을 위한 C, CUDA 등 라이브러리가 위치

2.1 파이토치 개요

▼ 그림 2-4 파이토치의 아키텍처



2.1 파이토치 개요

- **파이토치의 아키텍처**

  파이토치 API

  - 파이토치 API 계층에서는 사용자가 이해하기 쉬운 API를 제공하여 텐서에 대한 처리와 신경망을 구축하고 훈련할 수 있도록 도움
  - 이 계층에서는 사용자 인터페이스를 제공하지만 실제 계산은 수행하지 않음
  - 그 대신 C++로 작성된 파이토치 엔진으로 그 작업을 전달하는 역할만 함
  - 파이토치 API 계층에서는 사용자의 편의성을 위해 다음 패키지들이 제공

2.1 파이토치 개요

- **파이토치의 아키텍처**

  torch: GPU를 지원하는 텐서 패키지

  - 다차원 텐서를 기반으로 다양한 수학적 연산이 가능하도록 함
  - CPU뿐만 아니라 GPU에서 연산이 가능하므로 빠른 속도로 많은 양의 계산을 할 수 있음

2.1 파이토치 개요

- **파이토치의 아키텍처**

  torch.autograd: 자동 미분 패키지

  - Autograd는 텐서플로(TensorFlow), 카페(Caffe), CNTK 같은 다른 딥러닝 프레임워크와 가장 차별되는 패키지
  - 일반적으로 신경망에 사소한 변경(예 은닉층 노드 수 변경)이 있다면 신경망 구축을 처음부터 다시 시작해야 함
  - 파이토치는 '자동 미분(auto-differentiation)'이라고 하는 기술을 채택하여 미분 계산을 효율적으로 처리
  - 즉, '연산 그래프'가 즉시 계산(실시간으로 네트워크 수정이 반영된 계산)되기 때문에 사용자는 다양한 신경망을 적용해 볼 수 있음

2.1 파이토치 개요

- **파이토치의 아키텍처**

  torch.nn: 신경망 구축 및 훈련 패키지

  - torch.nn을 사용할 경우 신경망을 쉽게 구축하고 사용할 수 있음
  - 합성곱 신경망, 순환 신경망, 정규화 등이 포함되어 손쉽게 신경망을 구축하고 학습시킬 수 있음

  torch.multiprocessing: 파이썬 멀티프로세싱 패키지

  - 파이토치에서 사용하는 프로세스 전반에 걸쳐 텐서의 메모리 공유가 가능
  - 서로 다른 프로세스에서 동일한 데이터(텐서)에 대한 접근 및 사용이 가능

2.1 파이토치 개요

- **파이토치의 아키텍처**

  torch.utils: DataLoader 및 기타 유틸리티를 제공하는 패키지

  - 모델에 데이터를 제공하기 위한 torch.utils.data.DataLoader 모듈을 주로 사용
  - 병목 현상을 디버깅하기 위한 torch.utils.bottleneck, 모델 또는 모델의 일부를 검사하기 위한 torch.utils.checkpoint 등의 모듈도 있음

2.1 파이토치 개요

- **파이토치의 아키텍처**

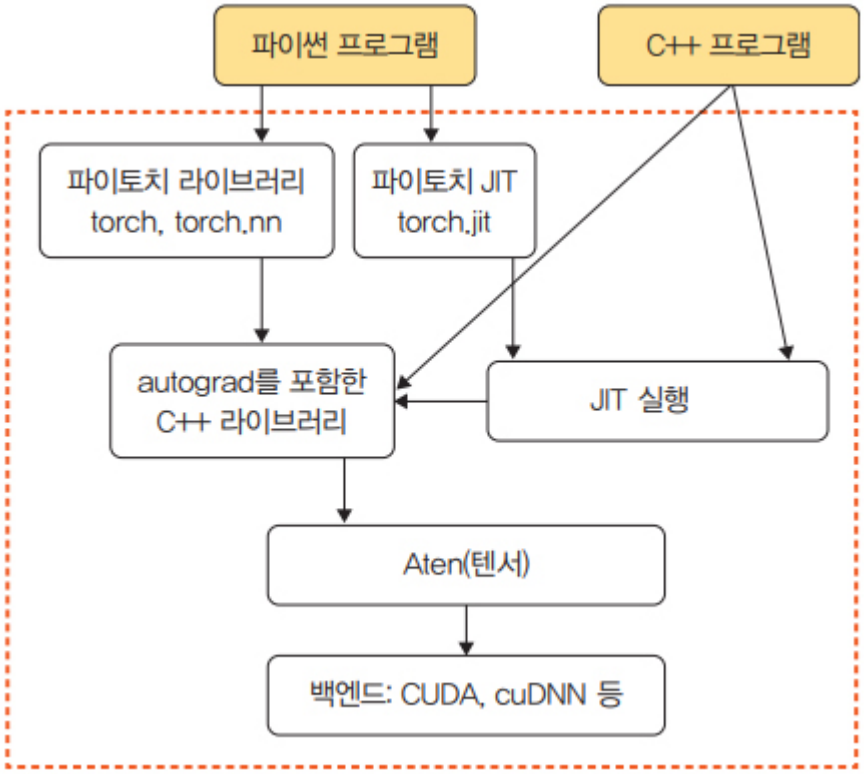
  파이토치 엔진

  - 파이토치 엔진은 Autograd C++, Aten C++, JIT C++, Python API로 구성
  - Autograd C++는 가중치, 바이어스를 업데이트하는 과정에서 필요한 미분을 자동으로 계산해 주는 역할
  - Aten C++는 C++ 텐서 라이브러리를 제공
  - JIT C++는 계산을 최적화하기 위한 JIT(Just In-Time) 컴파일러
  - 파이토치 엔진 라이브러리는 C++로 감싼(래핑(wrapping)) 다음 Python API 형태로 제공되기 때문에 사용자들이 손쉽게 모델을 구축하고 텐서를 사용할 수 있음

2.1 파이토치 개요

- **파이토치의 아키텍처**
  - 이들의 관계를 도식화하면 다음 그림과 같음

▼ **그림 2-5 파이토치 엔진**



2.1 파이토치 개요

- **파이토치의 아키텍처**

  연산 처리

  - 가장 아래 계층에 속하는 C 또는 CUDA 패키지는 상위의 API에서 할당된 거의 모든 계산을 수행
  - 여기에서 제공되는 패키지는 CPU와 GPU(TH(토치), THC(토치 CUDA))를 이용하여 효율적인 데이터 구조, 다차원 텐서에 대한 연산을 처리
  - 지금까지 파이토치의 아키텍처에 대한 전반적인 내용을 살펴보았음
  - 이제부터는 torch.tensor에 대해 알아보자
  - 계속 언급하고 있지만 파이토치에서는 텐서가 핵심

2.1 파이토치 개요

- **파이토치의 아키텍처**

  텐서를 메모리에 저장하기

  - 텐서는 그것이 1차원이든 N차원이든 메모리에 저장할 때는 1차원 배열 형태가 됨
  - 즉, 1차원 배열 형태여야만 메모리에 저장할 수 있음
  - 변환된 1차원 배열을 스토리지(storage)라고 함
  - 스토리지를 이해하기 위해서는 오프셋과 스트라이드 개념을 알아야 함
    - 오프셋(offset): 텐서에서 첫 번째 요소가 스토리지에 저장된 인덱스
    - 스트라이드(stride): 각 차원에 따라 다음 요소를 얻기 위해 건너뛰기(skip)가 필요한 스토리지의 요소 개수
    - 즉, 스트라이드는 메모리에서의 텐서 레이아웃을 표현하는 것으로 이해하면 됨
    - 요소가 연속적으로 저장되기 때문에 행 중심으로 스트라이드는 항상 1

2.1 파이토치 개요

- **파이토치의 아키텍처**
  - 왜 오프셋과 스트라이드라는 내용을 이해해야 할까?
  - 선형대수학을 배웠다면 전치 행렬이 무엇인지 알고 있을 것
  - 간단히 전치 행렬을 설명하면 다음과 같음
  - A 행렬에서 첫 번째 열을 첫 번째 행으로 위치시키고, 두 번째 열을 두 번째 행으로 위치시키며, A^T로 표현(동일한 원리를 텐서에 적용할 수 있음)

$$A=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} \Rightarrow A^T=\begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}$$

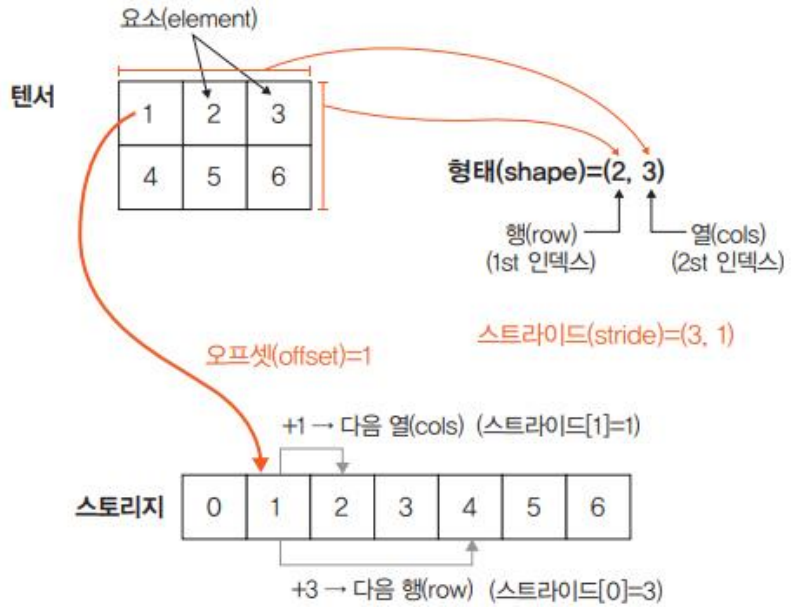
2.1 파이토치 개요

- **파이토치의 아키텍처**
  - 이것이 오프셋, 스트라이드와 무슨 관계가 있을까?
  - 조금 더 극적인 효과를 위해 행과 열의 수가 다른 텐서 A와 A^T를 생성해 보자
  - A와 A^T를 1차원 배열로 바꾸어서 메모리에 저장시키기 위해 텐서의 값들을 연속적으로 배치해 보자

▼ **그림 2-6 3차원 텐서를 1차원으로 변환**

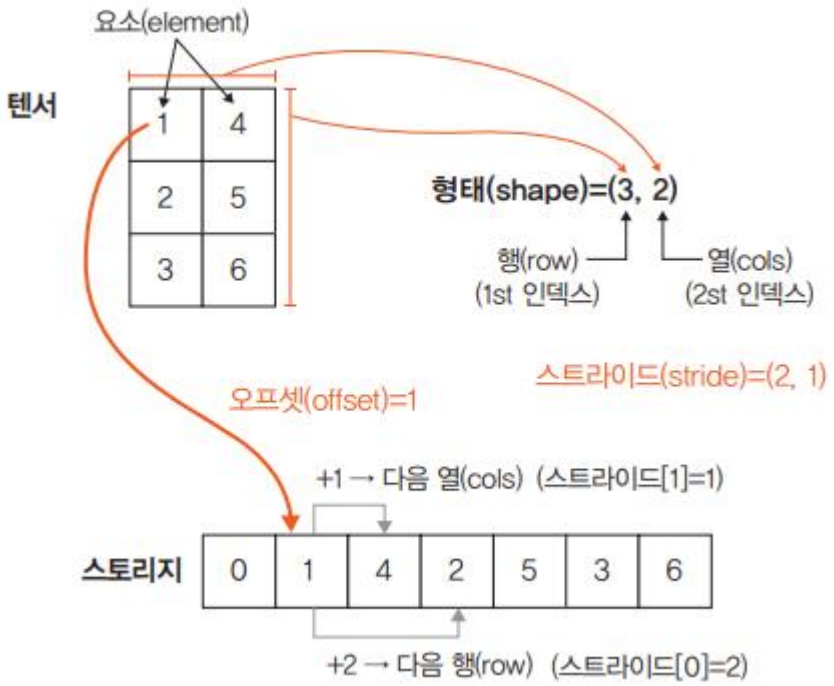
2.1 파이토치 개요

- **파이토치의 아키텍처**
  - A(2×3)와 A^T(3×2)는 다른 형태(shape)를 갖지만 스토리지의 값들은 서로 같음
  - A와 A^T를 구분하는 용도로 오프셋과 스트라이드를 사용
  - 다음 그림의 스토리지에서 2를 얻기 위해서는 1에서 1칸을 뛰어넘어야 하고, 4를 얻기 위해서는 3을 뛰어넘어야 함
  - 텐서에 대한 스토리지의 스트라이드는 (3, 1)



2.1 파이토치 개요

- **파이토치의 아키텍처**
  - 반면에 A의 전치 행렬은 좀 다름
  - 다음 그림의 스토리지에서 4를 얻기 위해서는 1에서 1칸을 뛰어넘어야 하고, 2을 얻기 위해서는 2를 뛰어넘어야 함
  - 텐서에 대한 스토리지의 스트라이드는 (2, 1)



2.2 파이토치 기초 문법

2.2 파이토치 기초 문법

- **텐서 다루기**

  텐서 생성 및 변환

  - 텐서는 파이토치의 가장 기본이 되는 데이터 구조
  - 넘파이의 ndarray와 비슷하며 GPU에서의 연산도 가능
  - 텐서 생성은 다음과 같은 코드를 이용

```
import torch
print(torch.tensor([[1,2],[3,4]])) ------ 2차원 형태의 텐서 생성
print(torch.tensor([[1,2],[3,4]], device="cuda:0")) ------ GPU에 텐서 생성
print(torch.tensor([[1,2],[3,4]], dtype=torch.float64)) ------ dtype을 이용하여 텐서 생성
```

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 다음은 생성된 텐서의 결과

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

tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 이번에는 텐서를 ndarray로 변환해 보자

```
temp = torch.tensor([[1,2],[3,4]])
print(temp.numpy()) ------ 텐서를 ndarray로 변환

temp = torch.tensor([[1,2],[3,4]], device="cuda:0")
print(temp.to("cpu").numpy()) ------ GPU상의 텐서를 CPU의 텐서로 변환한 후 ndarray로 변환
```

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 다음은 텐서를 ndarray로 변환한 결과

```
[[1 2]
 [3 4]]

[[1 2]
 [3 4]]
```

2.2 파이토치 기초 문법

- **텐서 다루기**

  텐서의 인덱스 조작

  - 텐서의 인덱스를 조작하는 방법은 여러 가지가 있음
  - 텐서는 넘파이의 ndarray를 조작하는 것과 유사하게 동작하기 때문에 배열처럼 인덱스를 바로 지정하거나 슬라이스 등을 사용할 수 있음
  - 텐서의 자료형은 다음과 같음

    - **torch.FloatTensor**: 32비트의 부동 소수점
    - **torch.DoubleTensor**: 64비트의 부동 소수점
    - **torch.LongTensor**: 64비트의 부호가 있는 정수

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 이외에도 다양한 유형의 텐서가 있음
  - 텐서의 인덱스 조작은 다음과 같은 코드를 이용

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

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 코드에서 -1번 인덱스는 맨 뒤에서부터 시작하는 인덱스를 의미
  - 다음은 인덱스 조작에 대한 결과

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

2.2 파이토치 기초 문법

- **텐서 다루기**

  텐서 연산 및 차원 조작

  - 텐서는 넘파이의 ndarray처럼 다양한 수학 연산이 가능하며, GPU를 사용하면 더 빠르게 연산할 수 있음
  - 참고로 텐서 간의 타입이 다르면 연산이 불가능함
  - 예를 들어 FloatTensor와 DoubleTensor 간에 사칙 연산을 수행하면 오류가 발생
  - 다음과 같이 벡터 두 개를 생성하여 사칙 연산을 할 수 있음

```
v = torch.tensor([1, 2, 3]) ------ 길이가 3인 벡터 생성
w = torch.tensor([3, 4, 6])
print(w - v) ------ 길이가 같은 벡터 간 뺄셈 연산
```

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 다음은 벡터 간 뺄셈 연산에 대한 결과

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

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 이번에는 텐서의 차원을 조작해 보자
  - 텐서의 차원에 대한 문제는 신경망에서 자주 다루어지므로 상당히 중요함
  - 텐서의 차원을 변경하는 가장 대표적인 방법은 view를 이용하는 것
  - 이외에도 텐서를 결합하는 stack, cat과 차원을 교환하는 t, transpose도 사용
  - view는 넘파이의 reshape과 유사하며 cat은 다른 길이의 텐서를 하나로 병합할 때 사용
  - 또한, transpose는 행렬의 전치 외에도 차원의 순서를 변경할 때도 사용

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 텐서의 차원을 조작하는 코드는 다음과 같음

```
temp = torch.tensor([
    [1, 2], [3, 4]]) ------ 2×2 행렬 생성

print(temp.shape)
print('----------------------')
print(temp.view(4, 1)) ------ 2×2 행렬을 4×1로 변형
print('----------------------')
print(temp.view(-1)) ------ 2×2 행렬을 1차원 벡터로 변형
print('----------------------')
print(temp.view(1, -1)) ---------- -1은 (1, ?)와 같은 의미로 다른 차원으로부터 해당 값을 유추하겠다는 것입니다. temp의 원소 개수(2×2=4)를 유지한 채 (1, ?)의 형태를 만족해야 하므로 (1, 4)가 됩니다.
print('----------------------')
print(temp.view(-1, 1)) ------ 앞에서와 마찬가지로 (?, 1)의 의미로 temp의 원소 개수(2×2=4)를 유지한 채 (?, 1)의 형태를 만족해야 하므로 (4, 1)이 됩니다.
```

2.2 파이토치 기초 문법

- **텐서 다루기**
  - 다음은 텐서의 차원을 조작한 결과

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

2.2 파이토치 기초 문법

- **데이터 준비**
  - 데이터 호출에는 파이썬 라이브러리(판다스(Pandas))를 이용하는 방법과 파이토치에서 제공하는 데이터를 이용하는 방법이 있음
  - 데이터가 이미지일 경우(이미지 모델을 사용해야 할 경우) 분산된 파일에서 데이터를 읽은 후 전처리를 하고 배치 단위로 분할하여 처리
  - 데이터가 텍스트일 경우(텍스트 모델을 사용해야 할 경우) 임베딩 과정을 거쳐 서로 다른 길이의 시퀀스(sequence)를 배치 단위로 분할하여 처리

2.2 파이토치 기초 문법

- **데이터 준비**
  - 다음은 파이토치를 이용하여 데이터셋을 불러오는 다양한 방법으로 각각의 방법을 하나씩 살펴보자

단순하게 파일을 불러와서 사용

- 판다스 라이브러리를 이용하여 JSON, PDF, CSV 등의 파일을 불러오는 방법
- 데이터가 복잡하지 않은 형태라면 단순하고 유용하게 사용될 수 있음
- 먼저 필요한 라이브러리를 설치
- 터미널 커맨드라인(아나콘다 프롬프트)에서 pip 명령어를 사용하여 다음 라이브러리를 설치

```
> pip install pandas
```

2.2 파이토치 기초 문법

- **데이터 준비**
  - 설치가 완료되었으면 예제 진행을 위한 라이브러리를 호출

```
import pandas as pd ------ pandas 라이브러리 호출
import torch ------ torch 라이브러리 호출
data = pd.read_csv('../class2.csv') ------ csv 파일을 불러옵니다.

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

CSV 파일의 x 칼럼의 값을 넘파이 배열로 받아 Tensor(dtype)으로 바꾸어 줍니다.

CSV 파일의 y 칼럼의 값을 넘파이 배열로 받아 Tensor(dtype)으로 바꾸어 줍니다.

2.2 파이토치 기초 문법

- **데이터 준비**

  커스텀 데이터셋을 만들어서 사용

  - 딥러닝은 기본적으로 대량의 데이터를 이용하여 모델을 학습
  - 데이터를 한 번에 메모리에 불러와서 훈련시키면 시간과 비용 측면에서 효율적이지 않음
  - 데이터를 한 번에 다 부르지 않고 조금씩 나누어 불러서 사용하는 방식이 커스텀 데이터셋(custom dataset)

2.2 파이토치 기초 문법

- **데이터 준비**
  - 먼저 CustomDataset 클래스를 구현하기 위해서는 다음 형태를 취해야 함

```
class CustomDataset(torch.utils.data.Dataset):
  def __init__(self): ------ 필요한 변수를 선언하고, 데이터셋의 전처리를 해 주는 함수
  def __len__(self): ------ 데이터셋의 길이. 즉, 총 샘플의 수를 가져오는 함수
  def __getitem__(self, index): ------ 데이터셋에서 특정 데이터를 가져오는 함수(index번째 데이터를 반환하는
                                       함수이며, 이때 반환되는 값은 텐서의 형태를 취해야 합니다)
```

2.2 파이토치 기초 문법

- **데이터 준비**
  - 커스텀 데이터셋 구현 방법에 대해 예제를 통해 구체적으로 알아보자

```
import pandas as pd
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

class CustomDataset(Dataset):
    def __init__(self, csv_file): ------ csv_file 파라미터를 통해 데이터셋을 불러옵니다.
        self.label = pd.read_csv(csv_file)

    def __len__(self): ------ 전체 데이터셋의 크기(size)를 반환합니다.
        return len(self.label)

    def __getitem__(self, idx): ------ 전체 x와 y 데이터 중에 해당 idx번째의 데이터를 가져옵니다.
        sample = torch.tensor(self.label.iloc[idx,0:3]).int()
        label = torch.tensor(self.label.iloc[idx,3]).int()
        return sample, label
```

2.2 파이토치 기초 문법

● 데이터 준비

```
tensor_dataset = CustomDataset('../covtype.csv') ------ 데이터셋으로 covtype.csv를 사용합니다.
dataset = DataLoader(tensor_dataset, batch_size=4, shuffle=True) ------
                                    데이터셋을 torch.utils.data.DataLoader에 파라미터로 전달합니다.
```

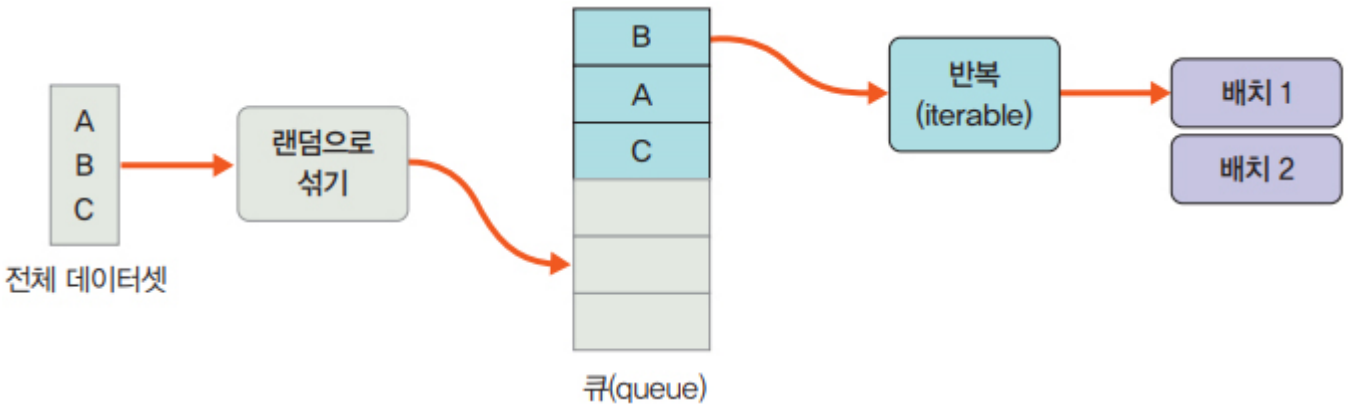
2.2 파이토치 기초 문법

- **데이터 준비**

  torch.utils.data.DataLoader

  - 데이터로더(DataLoader) 객체는 학습에 사용될 데이터 전체를 보관했다가 모델 학습을 할 때 배치 크기만큼 데이터를 꺼내서 사용
  - 이때 주의할 것은 데이터를 미리 잘라 놓는 것이 아니라 내부적으로 반복자(iterator)에 포함된 인덱스(index)를 이용하여 배치 크기만큼 데이터를 반환한다는 것

▼ **그림 2-9 데이터로더**



2.2 파이토치 기초 문법

- **데이터 준비**
  - 데이터로더는 다음과 같이 for 문을 이용하여 구문을 반복 실행하는 것과 같음

```
for i, data in enumerate(dataset,0):
    print(i, end='')
    batch=data[0]
    print(batch.size())
```

  - 다음과 같은 결과가 출력

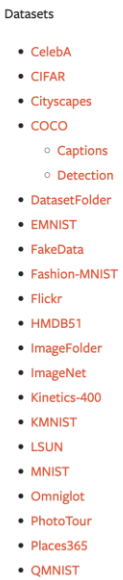
```
0torch.Size([4, 3])
1torch.Size([4, 3])
2torch.Size([4, 3])
3torch.Size([4, 3])
4torch.Size([3, 3])
```

2.2 파이토치 기초 문법

- **데이터 준비**

  파이토치에서 제공하는 데이터셋 사용

  - 토치비전(torchvision)은 파이토치에서 제공하는 데이터셋들이 모여 있는 패키지
  - MNIST, ImageNet을 포함한 유명한 데이터셋들을 제공하고 있음
  - 다음 URL에서 파이토치에서 제공하는 데이터셋을 확인할 수 있음
    - https://pytorch.org/vision/0.8/datasets.html



Datasets

- CelebA
- CIFAR
- Cityscapes
- COCO
  - Captions
  - Detection
- DatasetFolder
- EMNIST
- FakeData
- Fashion-MNIST
- Flickr
- HMDB51
- ImageFolder
- ImageNet
- Kinetics-400
- KMNIST
- LSUN
- MNIST
- Omniglot
- PhotoTour
- Places365
- QMNIST

2.2 파이토치 기초 문법

- **데이터 준비**
  - 파이토치에서 제공하는 데이터셋을 내려받으려면 먼저 requests 라이브러리를 설치
  - requests는 HTTP 요청에 대한 처리를 위해 사용하며, 기본 내장 모듈이 아니기 때문에 필요하다면 별도로 설치

```
> pip install requests
```

2.2 파이토치 기초 문법

- **데이터 준비**
  - 다음은 MNIST 데이터셋을 내려받는 예제

```
import torchvision.transforms as transforms

mnist_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (1.0,))
]) ------ 평균이 0.5, 표준편차가 1.0이 되도록 데이터의 분포(normalize)를 조정

from torchvision.datasets import MNIST
import requests
download_root = '../chap02/data/MNIST_DATASET' ------ 내려받을 경로 지정

train_dataset = MNIST(download_root, transform=mnist_transform, train=True,
                      download=True) ------ 훈련(training) 데이터셋
valid_dataset = MNIST(download_root, transform=mnist_transform, train=False,
                      download=True) ------ 검증(validation) 데이터셋
test_dataset = MNIST(download_root, transform=mnist_transform, train=False,
                     download=True) ------ 테스트(test) 데이터셋
```

2.2 파이토치 기초 문법

- **데이터 준비**
  - 코드를 실행하면 다음과 같이 출력되면서 데이터셋을 내려받음

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../chap02/
data/MNIST_DATASET\MNIST\raw\train-images-idx3-ubyte.gz
100.0%
Extracting ../chap02/data/MNIST_DATASET\MNIST\raw\train-images-idx3-ubyte.gz to ../
chap02/data/MNIST_DATASET\MNIST\raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../chap02/
data/MNIST_DATASET\MNIST\raw\train-labels-idx1-ubyte.gz
102.8%
Extracting ../chap02/data/MNIST_DATASET\MNIST\raw\train-labels-idx1-ubyte.gz to ../
chap02/data/MNIST_DATASET\MNIST\raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../chap02/
data/MNIST_DATASET\MNIST\raw\t10k-images-idx3-ubyte.gz
100.0%
Extracting ../chap02/data/MNIST_DATASET\MNIST\raw\t10k-images-idx3-ubyte.gz to ../
chap02/data/MNIST_DATASET\MNIST\raw
```

2.2 파이토치 기초 문법

- **데이터 준비**

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ../chap02/
data/MNIST_DATASET\MNIST\raw\t10k-labels-idx1-ubyte.gz
112.7%
e:\Anaconda3\envs\pytorch\lib\site-packages\torchvision\datasets\mnist.py:479:
UserWarning: The given NumPy array is not writeable, and PyTorch does not support
non-writeable tensors. This means you can write to the underlying (supposedly non-
writeable) NumPy array using the tensor. You may want to copy the array to protect its
data or make it writeable before converting it to a tensor. This type of warning will
be suppressed for the rest of this program. (Triggered internally at  ..\torch\csrc\
utils\tensor_numpy.cpp:143.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
Extracting ../chap02/data/MNIST_DATASET\MNIST\raw\t10k-labels-idx1-ubyte.gz to ../
chap02/data/MNIST_DATASET\MNIST\raw
Processing...
Done!
```

2.2 파이토치 기초 문법

- **모델 정의**
  - 파이토치에서 모델을 정의하기 위해서는 모듈(module)을 상속한 클래스를 사용
  - 그렇다면 모델과 모듈은 무엇이 다를까?

    - **계층**(layer): 모듈 또는 모듈을 구성하는 한 개의 계층으로 합성곱층(convolutional layer), 선형 계층(linear layer) 등이 있음
    - **모듈**(module): 한 개 이상의 계층이 모여서 구성된 것으로, 모듈이 모여 새로운 모듈을 만들 수도 있음
    - **모델**(model): 최종적으로 원하는 네트워크로, 한 개의 모듈이 모델이 될 수도 있음

2.2 파이토치 기초 문법

- **모델 정의**

  단순 신경망을 정의하는 방법

  - nn.Module을 상속받지 않는 매우 단순한 모델을 만들 때 사용
  - 구현이 쉽고 단순하다는 장점

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

2.2 파이토치 기초 문법

- **모델 정의**

  nn.Module()을 상속하여 정의하는 방법

  - 파이토치에서 nn.Module을 상속받는 모델은 기본적으로 _ _init_ _()과 forward() 함수를 포함
  - _ _init_ _()에서는 모델에서 사용될 모듈(nn.Linear, nn.Conv2d), 활성화 함수 등을 정의하고, forward() 함수에서는 모델에서 실행되어야 하는 연산을 정의

2.2 파이토치 기초 문법

- **모델 정의**
  - 다음은 파이토치에서 모델을 정의하는 코드

```
class MLP(Module):
    def __init__(self, inputs):
        super(MLP, self).__init__()
        self.layer = Linear(inputs, 1) ------ 계층 정의
        self.activation = Sigmoid() ------ 활성화 함수 정의

    def forward(self, X):
        X = self.layer(X)
        X = self.activation(X)
        return X
```

2.2 파이토치 기초 문법

- **모델 정의**

  Sequential 신경망을 정의하는 방법

  - nn.Sequential을 사용하면 _ _init_ _()에서 사용할 네트워크 모델들을 정의해 줄 뿐만 아니라 forward() 함수에서는 모델에서 실행되어야 할 계산을 좀 더 가독성이 뛰어나게 코드로 작성할 수 있음
  - Sequential 객체는 그 안에 포함된 각 모듈을 순차적으로 실행해 주는데 다음과 같이 코드를 작성할 수 있음

```
import torch.nn as nn
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2))
```

2.2 파이토치 기초 문법

- 모델 정의

```
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=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)
```

2.2 파이토치 기초 문법

- 모델 정의

```
            x = self.layer3(x)
            return x
model = MLP() ------ 모델에 대한 객체 생성

print("Printing children\n-------------------------------")
print(list(model.children()))
print("\n\nPrinting Modules\n-------------------------------")
print(list(model.modules()))
```

2.2 파이토치 기초 문법

- **모델 정의**
  - 이 코드를 실행하면 다음과 같이 출력

```
Printing children
------------------------------
[Sequential(
  (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
), Sequential(
  (0): Conv2d(64, 30, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
), Sequential(
  (0): Linear(in_features=750, out_features=10, bias=True)
  (1): ReLU(inplace=True)
)]
```

2.2 파이토치 기초 문법

- 모델 정의

```
Printing Modules
------------------------------
[MLP(
  (layer1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(64, 30, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Linear(in_features=750, out_features=10, bias=True)
    (1): ReLU(inplace=True)
```

2.2 파이토치 기초 문법

- **모델 정의**

```
    )
), Sequential(
  (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
), Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1)), ReLU(inplace=True),
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False),
Sequential(
  (0): Conv2d(64, 30, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
), Conv2d(64, 30, kernel_size=(5, 5), stride=(1, 1)), ReLU(inplace=True),
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False),
Sequential(
  (0): Linear(in_features=750, out_features=10, bias=True)
  (1): ReLU(inplace=True)
), Linear(in_features=750, out_features=10, bias=True), ReLU(inplace=True)]
```

2.2 파이토치 기초 문법

- **모델 정의**
  - nn.Sequential은 모델의 계층이 복잡할수록 효과가 뛰어남

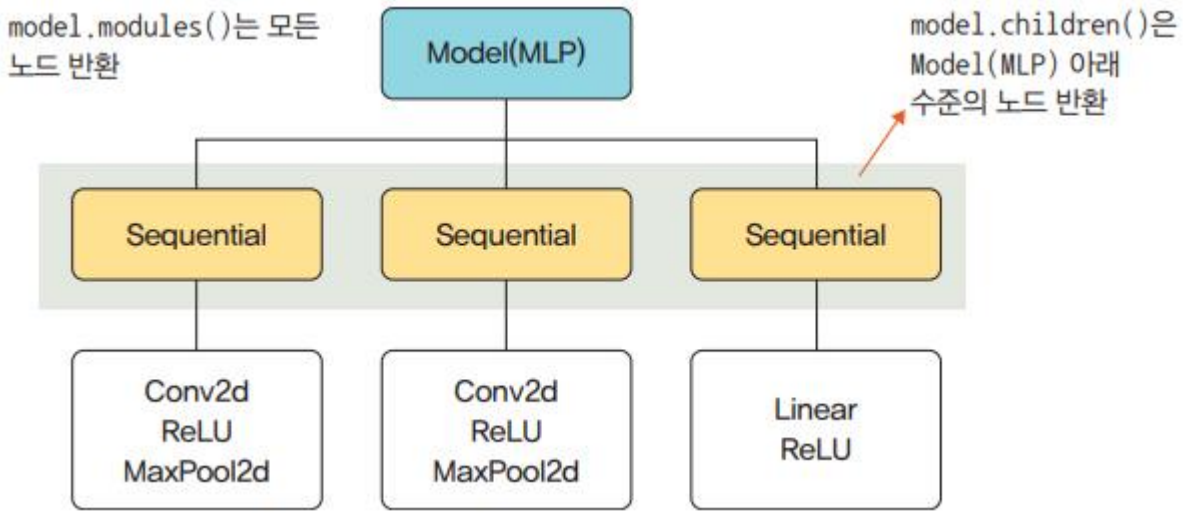
2.2 파이토치 기초 문법

- **모델 정의**

  model.modules() & model.children()

  - model.modules()는 모델의 네트워크에 대한 모든 노드를 반환하며, model.children()은 같은 수준(level)의 하위 노드를 반환

▼ **그림 2-10 model.modules( ) & model.children(**



2.2 파이토치 기초 문법

- **모델 정의**

  함수로 신경망을 정의하는 방법

  - Sequential을 이용하는 것과 동일하지만, 함수로 선언할 경우 변수에 저장해 놓은 계층들을 재사용할 수 있는 장점이 있음
  - 모델이 복잡해지는 단점도 있음

2.2 파이토치 기초 문법

- **모델 정의**
  - 참고로 복잡한 모델의 경우에는 함수를 이용하는 것보다는 nn.Module()을 상속받아 사용하는 것이 편리함

```
def MLP(in_features=1, hidden_features=20, out_features=1):
    hidden = nn.Linear(in_features=in_features, out_features=hidden_features,
                       bias=True)
    activation = nn.ReLU()
    output = nn.Linear(in_features=hidden_features, out_features=out_features,
                       bias=True)
    net = nn.Sequential(hidden, activation, output)
    return net
```

  - ReLU, Softmax 및 Sigmoid와 같은 활성화 함수는 모델을 정의할 때 지정

2.2 파이토치 기초 문법

- **모델의 파라미터 정의**
  - 모델을 학습하기 전에 필요한 파라미터들을 정의
  - 사전에 정의할 파라미터는 다음과 같음
    - **손실 함수**(loss function): 학습하는 동안 출력과 실제 값(정답) 사이의 오차를 측정
    - 즉, wx + b를 계산한 값과 실제 값인 y의 오차를 구해서 모델의 정확성을 측정
    - 손실 함수로 많이 사용되는 것은 다음과 같음
      - BCELoss: 이진 분류를 위해 사용
      - CrossEntropyLoss: 다중 클래스 분류를 위해 사용
      - MSELoss: 회귀 모델에서 사용

2.2 파이토치 기초 문법

- **모델의 파라미터 정의**
  - **옵티마이저**(optimizer): 데이터와 손실 함수를 바탕으로 모델의 업데이트 방법을 결정
  - 다음은 옵티마이저의 주요 특성

    - optimizer는 step() 메서드를 통해 전달받은 파라미터를 업데이트
    - 모델의 파라미터별로 다른 기준(예: 학습률)을 적용시킬 수 있음
    - torch.optim.Optimizer(params, defaults)는 모든 옵티마이저의 기본이 되는 클래스
    - zero_grad() 메서드는 옵티마이저에 사용된 파라미터들의 기울기(gradient)를 0으로 만듦
    - torch.optim.lr_scheduler는 에포크에 따라 학습률을 조절할 수 있음

2.2 파이토치 기초 문법

- **모델의 파라미터 정의**
  - 옵티마이저에 사용되는 종류는 다음과 같음
    - optim.Adadelta, optim.Adagrad, optim.Adam, optim.SparseAdam,
    - optim.Adamax
    - optim.ASGD, optim.LBFGS
    - optim.RMSProp, optim.Rprop, optim.SGD

2.2 파이토치 기초 문법

- **모델의 파라미터 정의**
  - **학습률 스케줄러**(learning rate scheduler): 미리 지정한 횟수의 에포크를 지날 때마다 학습률을 감소(decay)시켜 줌
    - 학습률 스케줄러를 이용하면 학습 초기에는 빠른 학습을 진행하다가 전역 최소점(global minimum) 근처에 다다르면 학습률을 줄여서 최적점을 찾아갈 수 있도록 해 줌
    - 학습률 스케줄러의 종류는 다음과 같음

      - optim.lr_scheduler.LambdaLR: 람다(lambda) 함수를 이용하여 그 함수의 결과를 학습률로 설정
      - optim.lr_scheduler.StepLR: 특정 단계(step)마다 학습률을 감마(gamma) 비율만큼 감소시킴
      - optim.lr_scheduler.MultiStepLR: StepLR과 비슷하지만 특정 단계가 아닌 지정된 에포크에만 감마 비율로 감소시킴
      - optim.lr_scheduler.ExponentialLR: 에포크마다 이전 학습률에 감마만큼 곱함
      - optim.lr_scheduler.CosineAnnealingLR: 학습률을 코사인(cosine) 함수의 형태처럼 변화시킴, 학습률이 커지기도 작아지기도 함
      - optim.lr_scheduler.ReduceLROnPlateau: 학습이 잘되고 있는지 아닌지에 따라 동적으로 학습률을 변화시킬 수 있음

2.2 파이토치 기초 문법

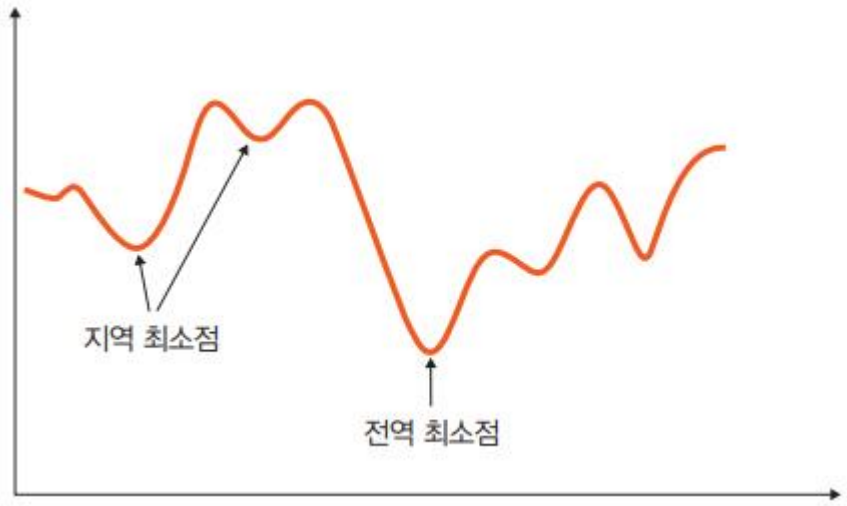
- **모델의 파라미터 정의**

  전역 최소점과 최적점

  - 손실 함수는 실제 값과 예측 값 차이를 수치화해 주는 함수
  - 이 오차 값이 클수록 손실 함수의 값이 크고, 오차 값이 작을수록 손실 함수의 값이 작아짐
  - 이 손실 함수의 값을 최소화하는 가중치와 바이어스를 찾는 것이 학습 목표
  - 전역 최소점(global minimum)은 오차가 가장 작을 때의 값을 의미하므로 우리가 최종적으로 찾고자 하는 것, 즉 최적점이라고 할 수 있음
  - 지역 최소점(local minimum)은 전역 최소점을 찾아가는 과정에서 만나는 홀(hole)과 같은 것으로 옵티마이저가 지역 최소점에서 학습을 멈추면 최솟값을 갖는 오차를 찾을 수 없는 문제가 발생

2.2 파이토치 기초 문법

▼ 그림 2-11 전역 최소점과 최적점



2.2 파이토치 기초 문법

- **모델의 파라미터 정의**
  - 다음은 모델의 파라미터를 정의하는 예시 코드

```
from torch.optim import optimizer
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)
for epoch in range(1, 100+1): ------ 에포크 수만큼 데이터를 반복하여 처리
    for x, y in dataloader: ------ 배치 크기만큼 데이터를 가져와서 학습 진행
        optimizer.zero_grad()
loss_fn(model(x), y).backward()
optimizer.step()
scheduler.step()
```

2.2 파이토치 기초 문법

- **모델 훈련**
  - 앞서 만들어 둔 데이터로 모델을 학습시킴
  - 이때 학습을 시킨다는 것은 y = wx + b라는 함수에서 w와 b의 적절한 값을 찾는다는 의미
  - w와 b에 임의의 값을 적용하여 시작하며 오차가 줄어들어 전역 최소점에 이를 때까지 파라미터(w, b)를 계속 수정

2.2 파이토치 기초 문법

- **모델 훈련**
  - 구체적으로 훈련 방법에 대해 알아보자
  - 가장 먼저 필요한 절차가 optimizer.zero_grad() 메서드를 이용하여 기울기를 초기화하는 것
  - 파이토치는 기울기 값을 계산하기 위해 loss.backward() 메서드를 이용하는데, 이것을 사용하면 새로운 기울기 값이 이전 기울기 값에 누적하여 계산
  - 이 방법은 순환 신경망(Recurrent Neural Network, RNN) 모델을 구현할 때 효과적이지만 누적 계산이 필요하지 않는 모델에 대해서는 불필요함
  - 기울기 값에 대해 누적 계산이 필요하지 않을 때는 입력 값을 모델에 적용하기 전에 optimizer.zero_grad() 메서드를 호출하여 미분 값(기울기를 구하는 과정에서 미분을 사용)이 누적되지 않게 초기화해 주어야 함

2.2 파이토치 기초 문법

▼ 그림 2-12 파이토치 학습 절차

딥러닝 학습 절차	파이토치 학습 절차
모델, 손실 함수, 옵티마이저 정의	모델, 손실 함수, 옵티마이저 정의
	optimizer.zero_grad(): 전방향 학습, 기울기 초기화
전방향 학습(입력 → 출력 계산)	output = model(input): 출력 계산
손실 함수로 출력과 정답의 차이(오차) 계산	loss = loss_fn(output, target): 오차 계산
역전파 학습(기울기 계산)	loss.backward(): 역전파 학습
기울기 업데이트	optimizer.step(): 기울기 업데이트

모델 학습 과정

2.2 파이토치 기초 문법

- **모델 훈련**
  - 다음은 loss.backward() 메서드를 이용하여 기울기를 자동 계산
  - loss.backward()는 배치가 반복될 때마다 오차가 중첩적으로 쌓이게 되므로 매번 zero_grad()를 사용하여 미분 값을 0으로 초기화
  - 다음은 모델을 훈련시키는 예시 코드

```
for epoch in range(100):
    yhat = model(x_train)
    loss = criterion(yhat, y_train)
    optimizer.zero_grad() ------ 오차가 중첩적으로 쌓이지 않도록 초기화
    loss.backward()
    optimizer.step()
```

2.2 파이토치 기초 문법

- **모델 평가**
  - 주어진 테스트 데이터셋을 사용하여 모델을 평가
  - 모델에 대한 평가는 함수와 모듈을 이용하는 두 가지 방법이 있음
  - 먼저 모델 평가를 위해 터미널 커맨드라인(아나콘다 프롬프트)에서 pip 명령어를 사용하여 다음 패키지를 설치

```
> pip install torchmetrics
```

2.2 파이토치 기초 문법

- **모델 평가**
  - 함수를 이용하여 모델을 평가하는 코드는 다음과 같음

```
import torch
import torchmetrics

preds = torch.randn(10, 5).softmax(dim=-1)
target = torch.randint(5, (10,))

acc = torchmetrics.functional.accuracy(preds, target)
```

모델을 평가하기 위해 torchmetrics.functional.accuracy 이용

2.2 파이토치 기초 문법

- **모델 평가**
  - 다음은 모듈을 이용하여 모델을 평가하는 코드

```
import torch
import torchmetrics
metric = torchmetrics.Accuracy() ------ 모델 평가(정확도) 초기화

n_batches = 10
for i in range(n_batches):
    preds = torch.randn(10, 5).softmax(dim=-1)
    target = torch.randint(5, (10,))

    acc = metric(preds, target)
    print(f"Accuracy on batch {i}: {acc}") ------ 현재 배치에서 모델 평가(정확도)

acc = metric.compute()
print(f"Accuracy on all data: {acc}") ------ 모든 배치에서 모델 평가(정확도)
```

2.2 파이토치 기초 문법

- **훈련 과정 모니터링**

  model.train() & model.eval()

  - model.train(): 훈련 데이터셋에 사용하며 모델 훈련이 진행될 것임을 알림
    이때 드롭아웃(dropout)이 활성화
  - model.eval(): 모델을 평가할 때는 모든 노드를 사용하겠다는 의미로 검증과 테스트
    데이터셋에 사용

2.2 파이토치 기초 문법

- **훈련 과정 모니터링**
  - model.train()과 model.eval()을 선언해야 모델의 정확도를 높일 수 있음
  - Model.train()은 앞에서 사용해 보았으니, 이번에는 model.eval()에 대한 사용 방법을 알아보자

```
model.eval() ------ 검증 모드로 전환(dropout=False)
with torch.no_grad(): ------①
    valid_loss = 0

    for x, y in valid_dataloader:
        outputs = model(x)
        loss = F.cross_entropy(outputs, y.long().squeeze())
        valid_loss += float(loss)
        y_hat += [outputs]

valid_loss = valid_loss / len(valid_loader)
```

2.2 파이토치 기초 문법

- **훈련 과정 모니터링**
  - ① model.eval()에서 with torch.no_grad()를 사용하는 이유는 다음과 같음
    - 파이토치는 모든 연산과 기울기 값을 저장
    - 검증(혹은 테스트) 과정에서는 역전파가 필요하지 않기 때문에 with torch.no_grad()를 사용하여 기울기 값을 저장하지 않도록 함
    - 이와 같은 과정을 통해 기울기 값을 저장하고 기록하는 데 필요한 메모리와 연산 시간을 줄일 수 있음

2.4 파이토치 코드 맛보기

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 데이터셋을 열어 보면 다음과 같이 특성(칼럼) 일곱 개로 구성되어 있음

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)**: 이 데이터는 unacc(허용 불가능한 수준) 및 acc(허용 가능한 수준), 양호(good) 및 매우 좋은(very good, vgood) 중 하나의 값을 갖음

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이때 **1**~ **6**의 칼럼 정보를 이용하여 일곱 번째 칼럼(차 상태)을 예측하는 코드를 구현해 보자
  - 먼저 필요한 라이브러리를 설치
  - 터미널 커맨드라인(아나콘다 프롬프트)에서 pip 명령어를 사용하여 다음 라이브러리를 설치

```
> pip install matplotlib
> pip install seaborn
> pip install scikit-learn
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

  설치한 라이브러리 설명

  1. **matplotlib**: 수많은 파이썬 라이브러리 중에서 2D, 3D 형태의 플롯(그래프)을 그릴 때 주로 사용하는 패키지(모듈)

  2. **seaborn**: 데이터 프레임으로 다양한 통계 지표를 표현할 수 있는 시각화 차트를 제공하기 때문에 데이터 분석에 활발히 사용되는 라이브러리

  3. **scikit-learn**: 분류(classification), 회귀(regression), 군집(clustering), 의사 결정 트리(decision tree) 등 다양한 머신 러닝 알고리즘을 적용할 수 있는 함수를 제공하는 머신 러닝 라이브러리

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이제 필요한 라이브러리(혹은 패키지)를 호출

코드 2-1 필요한 라이브러리 호출

```
import torch
import torch.nn as nn
import numpy as np ------ 벡터 및 행렬 연산에서 매우 편리한 기능을 제공하는 파이썬 라이브러리 패키지
import pandas as pd ------ 데이터 처리를 위해 널리 사용되는 파이썬 라이브러리 패키지
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 내려받은 데이터를 불러옴
  - 데이터는 예제 파일의 car_evaluation.csv
  - 데이터를 호출하기 위한 경로는 자신의 환경에 맞게 수정

코드 2-2 데이터 호출

```
dataset = pd.read_csv('../chap02/data/car_evaluation.csv') ------①
dataset.head() ------②
```

  - ① pd.read_csv() 메서드를 이용하여 ../chap02/data에 위치한 car_evaluation.csv 파일을 불러옴
  - ② 데이터프레임(DataFrame) 내의 처음 n줄을 출력해서 데이터의 내용을 확인할 수 있음
    - n의 기본값은 5
    - 이와 유사한 방법으로 데이터의 내용을 확인할 수 있는 것으로 dataset.tail()이 있음

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음은 dataset.head()에 대한 출력 결과
  - 참고로 인덱스는 0부터 시작

	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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 출력 결과 다섯 개의 행이 단어와 숫자로 구성되어 있는 것을 확인할 수 있음
  - 컴퓨터는 인간의 언어인 단어를 인식할 수 없기 때문에 단어를 벡터로 바꾸어 주는 임베딩(embedding) 처리가 필요함

2.4 파이토치 코드 맛보기

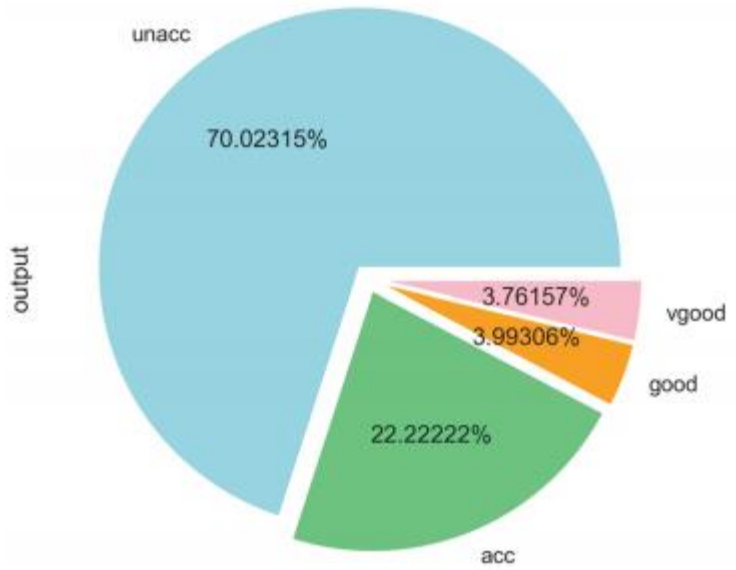
- **파이토치 코드 맛보기**
  - 주어진 데이터셋을 이해하기 쉽도록 분포 형태로 시각화하여 표현하면 다음과

코드 2-3 예제 데이터셋 분포

```
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))
```

2.4 파이토치 코드 맛보기

▼ 그림 2-26 예제 데이터셋 분포 결과



2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 결과에 따르면 대부분의 자동차(70%)는 허용 불가능한 상태에 있고 20%만 허용 가능한 수준
  - 즉, 양호한 상태의 자동차 비율이 매우 낮은 것을 볼 수 있음
  - 예제 데이터 정보를 확인했으니 본격적으로 데이터에 대한 전처리를 해 보자

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 딥러닝은 통계 알고리즘을 기반으로 하기 때문에 단어를 숫자(텐서)로 변환해야 함
  - 가장 먼저 필요한 전처리는 데이터를 파악하는 것
  - 주어진 데이터의 형태를 파악한 후 숫자로 변환해 주어야 하는데, 예제에서 다루는 데이터의 칼럼들은 모두 범주형 데이터(성별: 여자, 남자)로 구성되어 있음
  - 다음 코드로 단어를 배열로 변환하는 방법에 대해 간단히 살펴보자
  - 이 장의 코드는 맛보기 코드이므로 흐름만 간략히 익히고 넘어감

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이제 분석하기 좋게 데이터를 고치는 데이터 전처리(preprocessing)를 해야 함
  - astype() 메서드를 이용하여 범주 특성을 갖는 데이터를 범주형(category) 타입으로 변환
  - 파이토치를 이용한 모델 학습을 해야 하므로 범주형 타입을 텐서로 변환해야 함

코드 2-4 데이터를 범주형 타입으로 변환

```
categorical_columns = ['price', 'maint', 'doors', 'persons', 'lug_capacity', 'safety']

for category in categorical_columns:
    dataset[category] = dataset[category].astype('category')

price = dataset['price'].cat.codes.values ①
maint = dataset['maint'].cat.codes.values
doors = dataset['doors'].cat.codes.values
```

예제 데이터셋 칼럼들의 목록

astype() 메서드를 이용하여 데이터를 범주형으로 변환

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

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

categorical_data = np.stack([price, maint, doors, persons, lug_capacity, safety], 1) ------ ②
categorical_data[:10] ------ 합친 넘파이 배열 중 열 개의 행을 출력하여 보여 줍니다.
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - ① 범주형 데이터를 텐서로 변환하기 위해 다음과 같은 절차가 필요함

    범주형 데이터 → dataset[category] → 넘파이 배열(NumPy array) → 텐서(Tensor)

  - 즉, 파이토치로 모델을 학습시키기 위해서는 텐서 형태로 변환해야 하는데, 넘파이 배열을 통해 텐서를 생성할 수 있음

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 범주형 데이터(단어)를 숫자(넘파이 배열)로 변환하기 위해 cat.codes를 사용
  - cat.codes는 어떤 클래스가 어떤 숫자로 매핑되어 있는지 확인이 어려운 단점이 있으므로 주의해서 사용해야 함
  - ② np.stack은 두 개 이상의 넘파이 객체를 합칠 때 사용

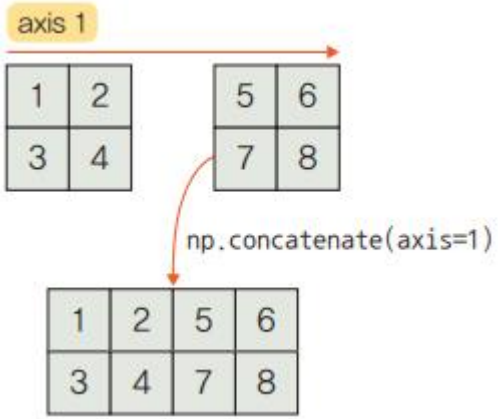
2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

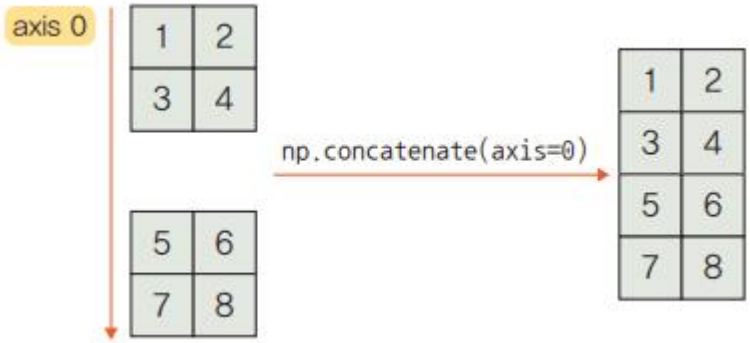
  np.stack과 np.concatenate

  - 넘파이 객체를 합칠 때 사용하는 메서드로는 np.stack과 np.concatenate가 있음
  - 이 두 메서드는 차원의 유지 여부에 대한 차이가 있음
  - np.concatenate는 다음 그림과 같이 선택한 축(axis)을 기준으로 두 개의 배열을 연결

▼ **그림 2-27 np.concatenate(axis=1)**



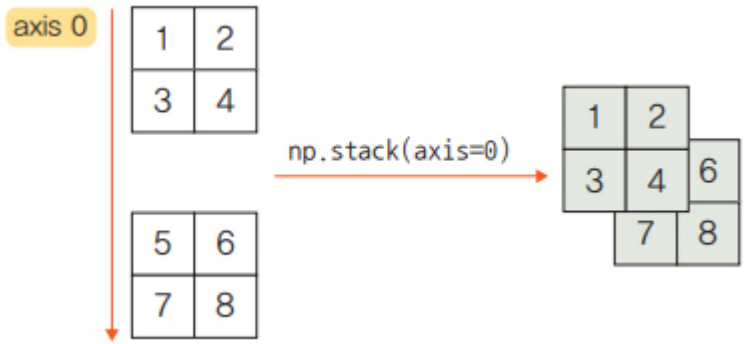
▼ **그림 2-28 np.concatenate(axis=0)**



2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - np.stack은 배열들을 새로운 축으로 합쳐 줌
  - 예를 들어 1차원 배열들을 합쳐서 2차원 배열을 만들거나 2차원 배열 여러 개를 합쳐 3차원 배열을 만듦
  - 반드시 두 배열의 차원이 동일해야 함

▼ **그림 2-30 np.stack(axis=0)**



2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 통해서 둘의 차이를 다시 살펴보자
  - 먼저 임의의 넘파이 배열 a, b, c를 정의
  - 이때 c는 다른 차원으로 정의
  - 이후 같은 차원을 갖는 a와 b에 대해 np.concatenate와 np.stack을 적용해 보자

```
a = np.array([[1, 2], [3, 4]]) ------ a.shape=(2, 2)
b = np.array([[5, 6], [7, 8]]) ------ b.shape=(2, 2)
c = np.array([[5, 6], [7, 8], [9, 10]]) ------ c.shape=(3, 2)

print(np.concatenate((a, b), axis=0)) ------ shape=(4, 2)
print('------------------------------')
print(np.stack((a, b), axis=0)) ------ shape=(2, 2, 2)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음은 np.concatenate와 np.stack을 적용한 결과
  - 차원이 같기 때문에 오류 없이 결과를 출력하고 있으며, np.stack의 경우에는 (2, 2, 2)로 차원이 변경된 것을 확인할 수 있음

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
------------------------------
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이번에는 서로 다른 차원을 합쳐 보자
  - 먼저 np.concatenate를 적용

```
print(np.concatenate((a, c), axis=0)) ------ shape=(5, 2)
```

  - 다음과 같이 출력

```
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]]
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이번에는 np.stack을 적용

```
print(np.stack((a, c), axis=0))
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - np.stack은 합치려는 두 넘파이 배열의 차원이 다르기 때문에 오류가 발생

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-d547630d1e7e> in <module>
----> 1 print(np.stack((a, c), axis=0))

<__array_function__ internals> in stack(*args, **kwargs)

e:\Anaconda3\envs\pytorch\lib\site-packages\numpy\core\shape_base.py in
stack(arrays, axis, out)
    425     shapes = {arr.shape for arr in arrays}
    426     if len(shapes) != 1:
--> 427         raise ValueError('all input arrays must have the same shape')
    428
    429     result_ndim = arrays[0].ndim + 1

ValueError: all input arrays must have the same shape
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음은 배열로 변환된 열 개의 행을 출력한 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)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이제 torch 모듈을 이용하여 배열을 텐서로 변환

코드 2-5 배열을 텐서로 변환

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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 텐서로 변환된 결과에 대한 열 개의 결과를 살펴보면 다음과 같음

```
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]])
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 마지막으로 레이블(outputs)로 사용할 칼럼에 대해서도 텐서로 변환해 줌
  - 이번에는 get_dummies를 이용하여 넘파이 배열로 변환

코드 2-6 레이블로 사용할 칼럼을 텐서로 변환

```
outputs = pd.get_dummies(dataset.output) ------ ①
outputs = outputs.values
outputs = torch.tensor(outputs).flatten() ------ 1차원 텐서로 변환

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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - ① get_dummies는 가변수(dummy variable)로 만들어 주는 함수
  - 가변수로 만들어 준다는 의미는 문자를 숫자 (0, 1)로 바꾸어 준다는 의미
  - 예를 위해 성별(gender), 몸무게(weight), 국적(nation)이라는 칼럼을 갖는 배열을 생성해 보자

```
import pandas as pd
import numpy as np

data = {
    'gender' : ['male','female','male'],
    'weight' : [72,55,68],
    'nation' : ['Japan','Korea','Australia']
}

df = pd.DataFrame(data)
df
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 생성된 배열의 형태는 다음과 같음

	gender	weight	nation
0	male	72	Japan
1	female	55	Korea
2	male	68	Australia

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 성별과 국적을 숫자로 변환하기 위해 get_dummies()를 적용

```
pd.get_dummies(df)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - et_dummies()를 적용한 결과는 다음과 같음
  - 원래 숫자의 값을 가졌던 몸무게는 변화가 없고 성별과 국적만 0과 1로 변경된 것을 확인할 수 있음

	weight	gender_female	gender_male	nation_Australia	nation_Japan	nation_Korea
0	72	0	1	0	1	0
1	55	1	0	0	0	1
2	68	0	1	1	0	0

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

  ravel(), reshape(), flatten()

  - ravel(), reshape(), flatten()은 텐서의 차원을 바꿀 때 사용
  - 이 메서드들은 다음과 같이 사용할 수 있음

```
a = np.array([[1, 2],
              [3, 4]])
print(a.ravel())
print(a.reshape(-1))
print(a.flatten())
```

  - 코드를 실행하면 다음과 같이 2차원 텐서가 1차원으로 변경되어 출력

```
[1 2 3 4]
[1 2 3 4]
[1 2 3 4]
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 실행하면 앞에서 텐서로 변환한 범주형 데이터와 레이블에 대한 형태가 출력

```
torch.Size([1728, 6])
torch.Size([6912])
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 워드 임베딩은 유사한 단어끼리 유사하게 인코딩되도록 표현하는 방법
  - 높은 차원의 임베딩일수록 단어 간의 세부적인 관계를 잘 파악할 수 있음
  - 단일 숫자로 변환된 넘파이 배열을 N차원으로 변경하여 사용
  - 배열을 N차원으로 변환하기 위해 먼저 모든 범주형 칼럼에 대한 임베딩 크기(벡터 차원)를 정의
  - 임베딩 크기에 대한 정확한 규칙은 없지만, 칼럼의 고유 값 수를 2로 나누는 것을 많이 사용
  - 예를 들어 price 칼럼은 네 개의 고유 값을 갖기 때문에 임베딩 크기는 4/2=2

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음 코드를 이용하여 (모든 범주형 칼럼의 고유 값 수, 차원의 크기) 형태로

코드 2-7 범주형 칼럼을 N차원으로 변환

```
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]
print(categorical_embedding_sizes)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음은 (모든 범주형 칼럼의 고유 값 수, 차원의 크기) 형태의 배열을 출력한 결과

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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 데이터셋을 훈련과 테스트 용도로 분리

코드 2-8 데이터셋 분리

```
total_records = 1728
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]
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 데이터를 훈련과 테스트 용도로 올바르게 분할했는지 확인하기 위해 레코드 개수를 출력해 보자

코드 2-9 데이터셋 분리 확인

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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음은 훈련 및 테스트 용도의 레코드 개수를 출력한 결과

```
1383
1383
345
345
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 데이터 준비가 끝났으므로, 모델의 네트워크를 생성

코드 2-10 모델의 네트워크 생성

```
class Model(nn.Module): ------①
    def __init__(self, embedding_size, output_size, layers, p=0.4): ------②
        super().__init__() ------③
        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
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

```
        for i in layers: ------④
            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 모든
                                                        계층에 대한 목록(all_layers)을 nn.Sequential
                                                        클래스로 전달
    def forward(self, x_categorical): ------⑤
        embeddings = []
        for i,e in enumerate(self.all_embeddings):
            embeddings.append(e(x_categorical[:,i]))
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

```
x = torch.cat(embeddings, 1) ------ 넘파이의 concatenate와 같지만 대상이 텐서가 됩니다.
x = self.embedding_dropout(x)
x = self.layers(x)
return x
```

2.4 파이토치 코드 맛보기

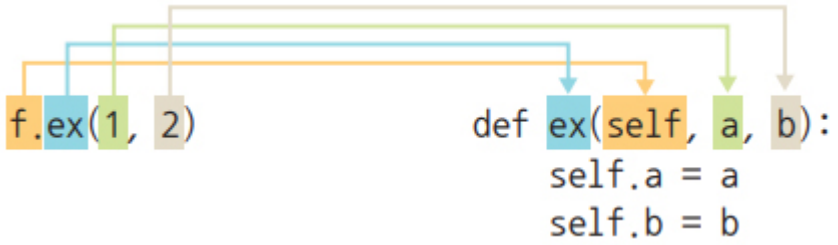
- **파이토치 코드 맛보기**
  - ① 클래스(class) 형태로 구현되는 모델은 nn.Module을 상속받음
  - ② _ _init_ _()은 모델에서 사용될 파라미터와 신경망을 초기화하기 위한 용도로 사용하며, 객체가 생성될 때 자동으로 호출
  - _ _init_ _()에서 전달되는 매개변수는 다음과 같음

```
def __init__(self, embedding_size, output_size, layers, p=0.4)
             ⓐ        ⓑ             ⓒ        ⓓ      ⓔ
```

ⓐ self: 첫 번째 파라미터는 self를 지정해야 하며 자기 자신을 의미
예를 들어 ex라는 함수가 있을 때 self 의미는 다음 그림과 같음

2.4 파이토치 코드 맛보기

▼ 그림 2-31 self 의미



2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

  ⓑ embedding_size: 범주형 칼럼의 임베딩 크기

  ⓒ output_size: 출력층의 크기

  ⓓ layers: 모든 계층에 대한 목록

  ⓔ p: 드롭아웃(기본값은 0.5)

- ③ super()._ _init_ _()은 부모 클래스(Model 클래스)에 접근할 때 사용하며 super는 self를 사용하지 않는 것에 주의해야 함

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - ④ 모델의 네트워크 계층을 구축하기 위해 for 문을 이용하여 각 계층을 all_layers 목록에 추가
  - 추가된 계층은 다음과 같음

    • Linear: 선형 계층(linear layer)은 입력 데이터에 선형 변환을 진행한 결과

    선형 변환을 위해서는 다음 수식을 사용

$$y = Wx + b$$

(y: 선형 계층의 출력 값, W: 가중치, x: 입력 값, b: 바이어스)

    선형 계층은 입력과 가중치를 곱한 후 바이어스를 더한 결과

    • ReLU: 활성화 함수로 사용

    • BatchNorm1d: 배치 정규화(batch normalization) 용도로 사용

    • Dropout: 과적합 방지에 사용

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - ⑤ forward() 함수는 학습 데이터를 입력받아서 연산을 진행
  - forward() 함수는 모델 객체를 데이터와 함께 호출하면 자동으로 실행

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 모델 훈련을 위해 앞에서 정의했던 Model 클래스의 객체를 생성
  - 객체를 생성하면서 (범주형 칼럼의 임베딩 크기, 출력 크기, 은닉층의 뉴런, 드롭아웃)을 전달
  - 여기에서는 은닉층을 [200,100,50]으로 정의했지만 다른 크기로 지정하여 테스트해 보는 것도 학습하는 데 도움이 될 것

코드 2-11 Model 클래스의 객체 생성

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

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 실행하면 모델에 대한 구조(네트워크)를 보여 줌

```
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_
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

```
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)
  )
)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 모델을 훈련시키기 전에 손실 함수와 옵티마이저에 대해 정의해야 함
  - 이번 예제는 데이터를 분류해야 하는 것으로 크로스 엔트로피(cross entropy) 손실 함수를 사용
  - 또한, 옵티마이저로는 아담(Adam)을 사용

코드 2-12 모델의 파라미터 정의

```
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 파이토치는 GPU에 최적화된 딥러닝 프레임워크
  - GPU가 없다면 CPU를 사용할 수 있도록 지정해 주어야 함
  - 다음은 GPU가 있다면 GPU를 사용하고, 없다면 CPU를 사용하도록 하는 코드

코드 2-13 CPU/GPU 사용 지정

```
if torch.cuda.is_available():
    device = torch.device('cuda') ------ GPU가 있다면 GPU를 사용
else:
    device = torch.device('cpu') ------ GPU가 없다면 CPU를 사용
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 모델 훈련에 필요한 모든 준비가 완료
  - 이제 준비된 데이터를 이용하여 모델을 학습시킴

코드 2-14 모델 학습

```
epochs = 500
aggregated_losses = []
train_outputs = train_outputs.to(device=device, dtype=torch.int64)
for i in range(epochs): ------ for 문은 500회 반복되며, 각 반복마다 손실 함수가 오차를 계산
    i += 1
    y_pred = model(categorical_train_data)
    single_loss = loss_function(y_pred, train_outputs)
    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}')
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

```
optimizer.zero_grad()
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 출력
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 실행하면 25 에포크마다 출력된 오차 정보를 보여 줌

```
epoch:   1 loss: 1.63872778
epoch:  26 loss: 1.46383297
epoch:  51 loss: 1.36062038
epoch:  76 loss: 1.25486016
epoch: 101 loss: 1.11357403
epoch: 126 loss: 0.94361728
epoch: 151 loss: 0.84047800
epoch: 176 loss: 0.74985331
epoch: 201 loss: 0.70034856
epoch: 226 loss: 0.65812957
epoch: 251 loss: 0.63274646
epoch: 276 loss: 0.61346799
epoch: 301 loss: 0.60412955
epoch: 326 loss: 0.59235305
epoch: 351 loss: 0.59519970
epoch: 376 loss: 0.57206368
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

```
epoch: 401 loss: 0.57828188
epoch: 426 loss: 0.58816069
epoch: 451 loss: 0.57712984
epoch: 476 loss: 0.57470286
epoch: 500 loss: 0.5725595951
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 학습이 끝났으므로 테스트 데이터셋으로 예측을 진행해 보자
  - 앞에서 준비했던 categorical_test_data 데이터셋을 모델에 적용

코드 2-15 테스트 데이터셋으로 모델 예측

```
test_outputs = test_outputs.to(device=device, dtype=torch.int64)
with torch.no_grad():
    y_val = model(categorical_test_data)
    loss = loss_function(y_val, test_outputs)
print(f'Loss: {loss:.8f}')
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 실행하면 테스트 용도의 데이터셋에 대한 손실 값을 보여 줌
  - 이 값은 훈련 데이터셋에서 도출된 손실 값과 비슷하므로 과적합은 발생하지 않았다고 판단할 수 있음

```
Loss: 0.55525565
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 이제 테스트 데이터셋을 이용했을 때 모델이 얼마나 잘 예측하는지 살펴보자
  - 앞에서 모델 네트워크의 output_size에 4를 지정
  - 즉, 출력층에 네 개의 뉴런이 포함되도록 지정했으므로 각 예측에는 네 개의 값이 포함될 것

코드 2-16 모델의 예측 확인

```
print(y_val[:5])
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 모델이 얼마나 잘 예측하는지 확인하기 위해 처음 다섯 개의 값을 출력하면 다음과 같음

```
tensor([[ 2.7215,  1.6601, -2.2784, -2.1693],
        [ 2.8467,  1.7512, -2.3375, -2.2104],
        [ 1.5050,  1.0164, -2.3425, -2.1092],
        [ 2.9343,  1.4001, -5.3671, -5.7565],
        [ 3.7045,  2.1831, -7.9224, -7.9769]])
```

  - 값이 출력되었지만 어떤 의미인지 이해하기 어려워 보임
  - 실제 출력이 0이면 인덱스 0(2.7215)의 값이 인덱스 1(1.6601)의 값보다 높아야 함

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 다음과 같은 코드를 이용하여 목록에서 가장 큰 값을 갖는 인덱스를 알아보자
  - 다시 말하지만 실제 값이 아닌 인덱스를 찾는 것

코드 2-17 가장 큰 값을 갖는 인덱스 확인

```
y_val = np.argmax(y_val, axis=1)
print(y_val[:5])
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - y_val에서 처음 다섯 개의 값이 출력

    ```
    tensor([0, 0, 0, 0, 0])
    ```

  - 출력 결과 모두 인덱스 0이 출력
  - 즉, 인덱스가 0인 값이 인덱스가 1인 값보다 크므로 처리된 출력이 0임을 확인할 수 있음

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 마지막으로 sklearn.metrics 모듈의 classification_report, confusion_matrix, accuracy_score 클래스를 사용하여 정확도, 정밀도와 재현율을 알아보자

코드 2-18 테스트 데이터셋을 이용한 정확도 확인

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
print(confusion_matrix(test_outputs,y_val))
print(classification_report(test_outputs,y_val))
print(accuracy_score(test_outputs, y_val))
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 코드를 실행하면 모델 평가를 실행한 결과가 출력

```
[[257   2]
 [ 84   2]]
              precision    recall  f1-score   support

           0       0.75      0.99      0.86       259
           1       0.50      0.02      0.04        86

    accuracy                           0.75       345
   macro avg       0.63      0.51      0.45       345
weighted avg       0.69      0.75      0.65       345

0.7507246376811594
```

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 신경망에서 필요한 모든 파라미터를 무작위로 선택했다는 것을 감안할 때 75%의 정확도는 나쁘지 않음
  - 파라미터(예 훈련/테스트 데이터셋 분할, 은닉층 개수 및 크기 등)를 변경하면서 더 나은 성능을 찾아보는 것도 학습에 도움이 될 것
  - 마지막으로 딥러닝 분류 모델의 성능 평가 지표를 알아보자
  - 성능 평가 지표로 정확도(accuracy), 재현율(recall), 정밀도(precision), F1-스코어(F1-score)가 있음

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**
  - 정확도를 확인하기 전에 필요한 용어들부터 살펴보자

    - **True Positive**: 모델(분류기)이 '1'이라고 예측했는데 실제 값도 '1'인 경우
    - **True Negative**: 모델(분류기)이 '0'이라고 예측했는데 실제 값도 '0'인 경우
    - **False Positive**: 모델(분류기)이 '1'이라고 예측했는데 실제 값은 '0'인 경우로, Type I 오류라고도 함
    - **False Negative**: 모델(분류기)이 '0'이라고 예측했는데 실제 값은 '1'인 경우로, Type II 오류라고도 함

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

정확도

- 전체 예측 건수에서 정답을 맞힌 건수의 비율
- 이때 맞힌 정답이 긍정(positive)이든 부정(negative)이든 상관없음

$$\frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

재현율

- 실제로 정답이 1이라고 할 때 모델(분류기)도 1로 예측한 비율
- 처음부터 데이터가 1일 확률이 적을 때 사용하면 좋음

$$\frac{True\ Positive}{True\ Positive + False\ Negative}$$

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

  정밀도

  - 모델(분류기)이 1이라고 예측한 것 중에서 실제로 정답이 1인 비율

$$\frac{True\ Positive}{True\ Positive + False\ Positive}$$

2.4 파이토치 코드 맛보기

- **파이토치 코드 맛보기**

  F1-스코어

  - 일반적으로 정밀도와 재현율은 트레이드오프(trade-off) 관계
  - 정밀도가 높으면 재현율이 낮고, 재현율이 높으면 정밀도가 낮음
  - 이러한 트레이드오프 문제를 해결하려고 정밀도와 재현율의 조화 평균(harmonic mean)을 이용한 것이 F1-스코어 평가
  - 이때 조화 평균은 다음 공식으로 구할 수 있음

$$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$