

chap01 - 02

실습 링크 :

<https://colab.research.google.com/drive/1AtGaD4zCX7aBKWjtg16DeHfvvhg-J2eV?usp=sharing>

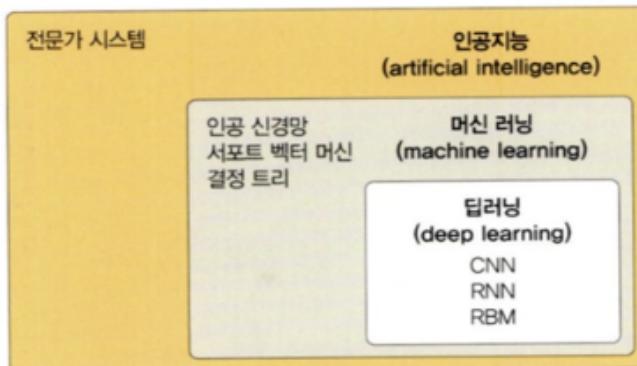
1.1 인공지능, 머신 러닝과 딥러닝

인공지능, 머신 러닝, 딥러닝의 관계

인공지능(AI)은 인간의 지능을 모방하여 컴퓨터가 스스로 판단·학습할 수 있도록 만드는 기술이다. AI를 구현하는 대표적인 방법이 **머신 러닝(Machine Learning)**이고, 그중에서도 인공신경망 기반 접근이 **딥러닝(Deep Learning)**이다.

아래 그림처럼 **AI** ⊃ **머신러닝** ⊃ **딥러닝** 관계로 정리할 수 있다.

▼ 그림 1-1 인공지능과 머신 러닝, 딥러닝의 관계

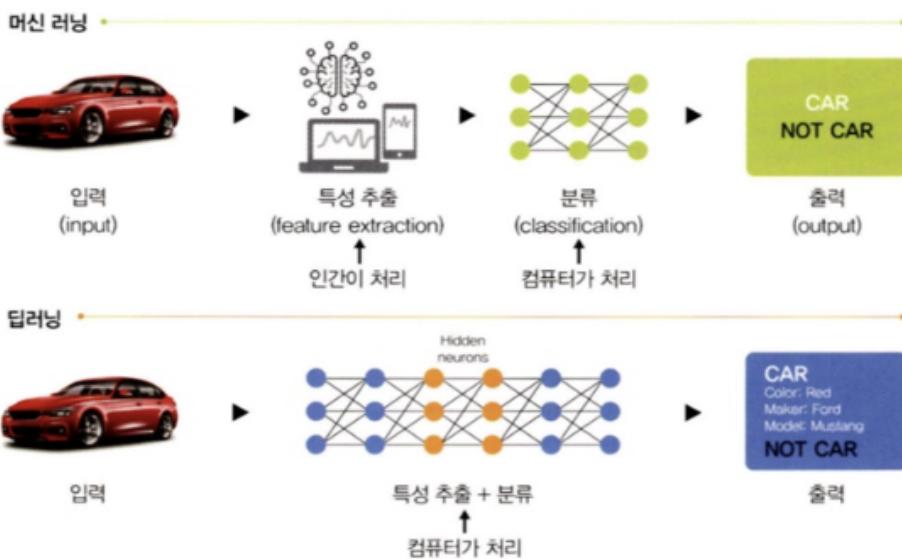


머신러닝은 주어진 데이터를 사람이 가공하여 특징을 뽑아내고 학습에 활용하지만,
딥러닝은 대규모 데이터를 입력받아 **신경망**이 스스로 특징을 추출하고 학습한다.

머신러닝 vs 딥러닝

책에서는 자동차 이미지를 예로 들어 두 방법의 차이를 설명한다.

▼ 그림 1-2 머신 러닝과 딥러닝 차이



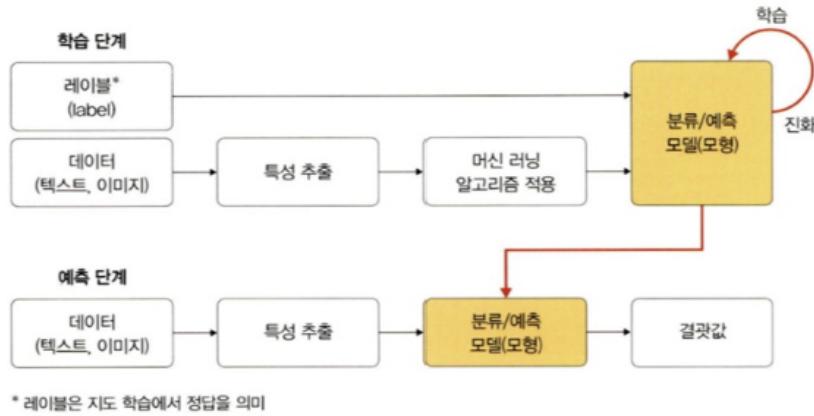
- **머신러닝:** 사람이 먼저 "차의 특징(바퀴, 창문, 색 등)"을 추출해주고, 이를 기반으로 알고리즘이 분류를 수행합니다.
- **딥러닝:** 원본 이미지를 그대로 입력받아 신경망이 특징 추출과 분류를 모두 처리합니다.

구분	머신러닝	딥러닝
특징 추출	사람이 직접 수행	신경망이 자동 수행
데이터 필요량	수천 개 정도	수백만 개 이상
훈련 시간	짧음	길음
결과	점수·숫자 중심	점수, 텍스트, 이미지 등 다양한 출력

1.2 머신 러닝이란?

머신러닝은 데이터에서 패턴을 찾아내고 예측하는 기술이다. 크게 두 단계로 나뉜다.

1. 학습 단계 (Learning): 레이블이 포함된 데이터를 통해 알고리즘이 규칙을 학습
2. 예측 단계 (Prediction): 새로운 데이터를 입력받아 학습된 모델로 예측 수행



머신러닝의 핵심 요소

머신러닝은 크게 두 가지 구성 요소로 설명된다.

1. **데이터:** 학습에 사용되는 자료. 보통 훈련 데이터셋(80%)과 테스트 데이터셋(20%)으로 나눔
2. **모델:** 데이터를 통해 학습한 규칙. 모델을 선택·평가·업데이트하는 과정을 반복

예: 오토바이 이미지 → 바퀴, 핸들 등 주요 특징 추출

모델 학습 절차는 아래와 같이 순환 구조를 가진다.

1. 모델 가설 선택
 2. 학습 및 평가
 3. 모델 업데이트
- 최적 모델 완성

데이터셋 분할

데이터셋은 일반적으로 훈련(training), 검증(validation), 테스트(test) 세 가지로 나눠진다.

- **훈련 데이터:** 모델 학습에 사용
- **검증 데이터:** 모델 성능을 중간 점검하고 튜닝
- **테스트 데이터:** 최종 성능 평가

머신러닝 학습 알고리즘

머신러닝 알고리즘은 크게 세 가지 범주로 나뉜다.

1. 지도 학습(Supervised Learning)

- 라벨이 있는 데이터 학습
- 분류(Classification), 회귀(Regression)
- 예: KNN, SVM, 의사결정트리, 로지스틱 회귀, 선형 회귀

2. 비지도 학습(Unsupervised Learning)

- 라벨 없는 데이터에서 패턴 찾기
- 군집화(Clustering), 차원 축소(Dimensionality Reduction)
- 예: K-means, DBSCAN, PCA

3. 강화 학습(Reinforcement Learning)

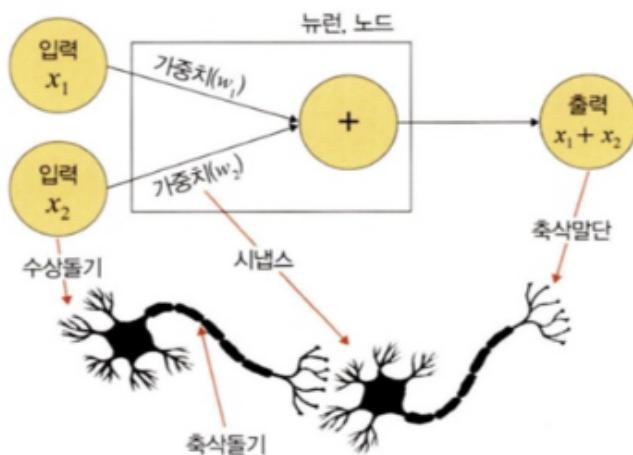
- 보상(Reward)을 통해 행동 학습
- 대표 알고리즘: 마르코프 결정 과정(MDP)
- 예: 쿠키런 게임에서 에이전트가 점프/슬라이드 선택

1.3 딥러닝이란?

인간의 뇌에서 영감을 받은 딥러닝

딥러닝은 [인간의 신경망 구조](#)를 모방한 [심층 신경망 이론](#)을 기반으로 한 머신러닝 방법이다.
뇌 속 수많은 뉴런(Neuron)과 시냅스(Synapse) 연결 방식을 본떠 컴퓨터에 적용한 것.

▼ 그림 1-10 인간의 신경망 원리를 모방한 심층 신경망



이런 구조를 컴퓨터 모델로 단순화한 것이 바로 인공신경망이다.

딥러닝 학습 과정

딥러닝의 학습 절차는 머신러닝과 크게 다르지 않지만, **신경망 모델 정의와 학습 과정**이 핵심이다.

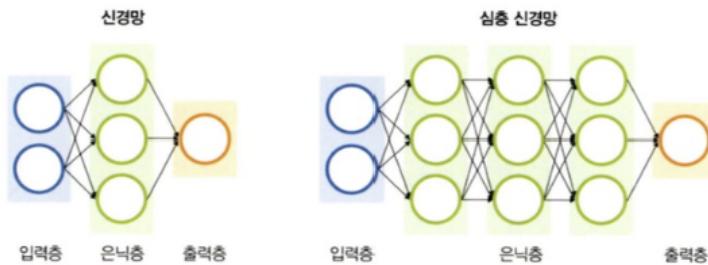
1. **데이터 준비**: 공개 데이터셋(Kaggle, PyTorch 튜토리얼 등)을 활용 가능
2. **모델 정의**: 신경망 구조(레이어, 노드 수)를 설계
3. **컴파일**: 활성 함수, 손실 함수, 옵티마이저 선택
4. **훈련(Training)**: 배치(batch) 단위로 데이터 학습
5. **예측(Prediction)**: 새로운 데이터 적용

특히 배치 크기와 에포크 수 같은 하이퍼파라미터 선택이 모델 성능에 큰 영향을 준다.

신경망과 심층 신경망

- **신경망**: 입력층-은닉층 1개-출력층 구조
- **심층 신경망(Deep Neural Network, DNN)**: 은닉층이 2개 이상인 구조

심층 신경망은 데이터의 중요한 특징을 **스스로 학습**할 수 있어 딥러닝이 머신러닝과 구분되는 핵심이다.

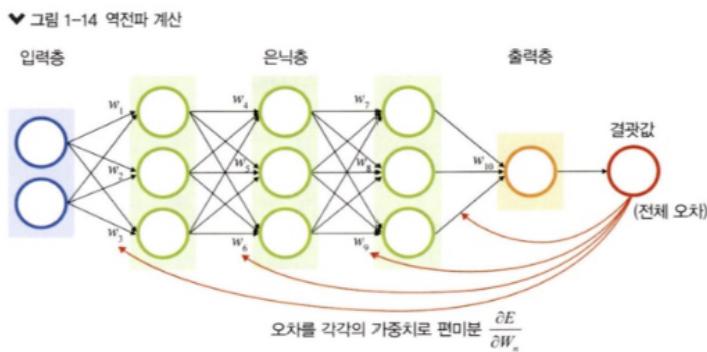


역전파 알고리즘

딥러닝 학습의 핵심은 **역전파(Backpropagation)**이다.

출력층의 오차를 가중치별로 분배하여 다시 입력층으로 전달하면서 가중치를 업데이트한다.

이 과정 덕분에 신경망이 스스로 더 정확한 예측을 하도록 점점 개선된다.



대표적인 딥러닝 알고리즘

딥러닝에서도 머신러닝과 마찬가지로 **지도 학습, 비지도 학습, 전이 학습, 강화 학습** 방식이 사용된다.

1. 지도 학습 (Supervised Learning)

- **CNN(합성곱 신경망)**: 이미지 인식·분류
- **RNN(순환 신경망)**: 시계열 데이터 처리, LSTM으로 발전
- 예시: 자율주행 자동차 이미지 인식, 주가 예측

2. 비지도 학습 (Unsupervised Learning)

- 대표적으로 **워드 임베딩(Word Embedding)** → 단어를 벡터로 변환
- Word2Vec, GloVe 등이 대표적

- 군집 알고리즘과 함께 사용 시 데이터 패턴을 효율적으로 탐색 가능

3. 전이 학습 (Transfer Learning)

- 사전 학습된 모델(Pre-trained model)을 가져와 일부만 조정
- 대표 모델: VGG, Inception, MobileNet
- 데이터가 부족한 상황에서도 효율적으로 학습 가능

4. 강화 학습 (Reinforcement Learning)

- 마르코프 결정 과정(MDP) 기반
- 보상에 따라 행동을 학습
- 게임 AI, 로보틱스, 추천 시스템에 활용

2.1 파이토치 개요

파이토치(PyTorch)는 2017년 초에 공개된 딥러닝 프레임워크로, 루아(Lua) 기반 토치(Torch)에서 발전한 버전이다. 현재는 텐서플로(TensorFlow)와 함께 가장 많이 쓰이는 프레임워크.

파이토치가 주목받는 가장 큰 이유는 간결하고 빠른 구현 덕분이다.

- **GPU 연산 지원:** 복잡한 연산을 CUDA 기반 GPU에서 빠르게 처리
- **유연한 연구 플랫폼:** 동적 신경망 구조를 쉽게 정의 가능
- **직관적인 코드:** 파이썬과 잘 통합되어 연구·실무에서 모두 활용

1. 파이토치의 핵심 요소

(1) GPU

딥러닝에서 자주 등장하는 **기울기 계산**을 빠르게 하기 위해 GPU는 필수적이다. CUDA와 cuDNN API를 통해 병렬 연산을 수행하며 CPU보다 훨씬 빠른 학습이 가능합니다.

(2) 텐서(Tensor)

텐서는 파이토치의 데이터 기본 단위이다.

- 벡터(1차원), 행렬(2차원), 텐서(3차원 이상)로 확장
- `.cuda()` 명령어로 GPU 연산 수행 가능
- `torch.FloatTensor` : 32비트 부동소수점
- `torch.DoubleTensor` : 64비트 부동소수점
- `torch.LongTensor` : 64비트 정수

예: 2차원 텐서 생성

```
import torch
x = torch.tensor([[1, 2], [3, 4]])
print(x)
```

출력:

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

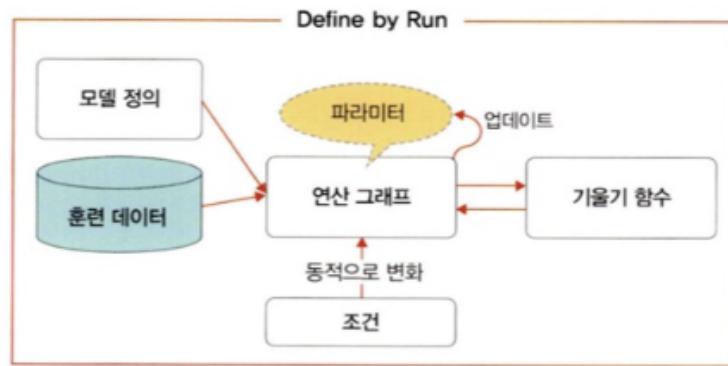
2. 동적 신경망 (Define by Run)

파이토치의 강점은 **Define by Run** 개념이다.

- 연산 그래프가 고정되어 있지 않고 실행할 때마다 동적으로 변함
- 조건문, 반복문 등을 모델에 자유롭게 적용 가능

이 덕분에 파이토치는 직관적이고 디버깅이 쉬운 프레임워크로 자리잡았음.

"*Define by Run*" 구조는 PyTorch가 어떻게 파라미터, 연산 그래프, 조건, 손실 함수를 유연하게 연결하는지 보여다.



Define by Run 구조

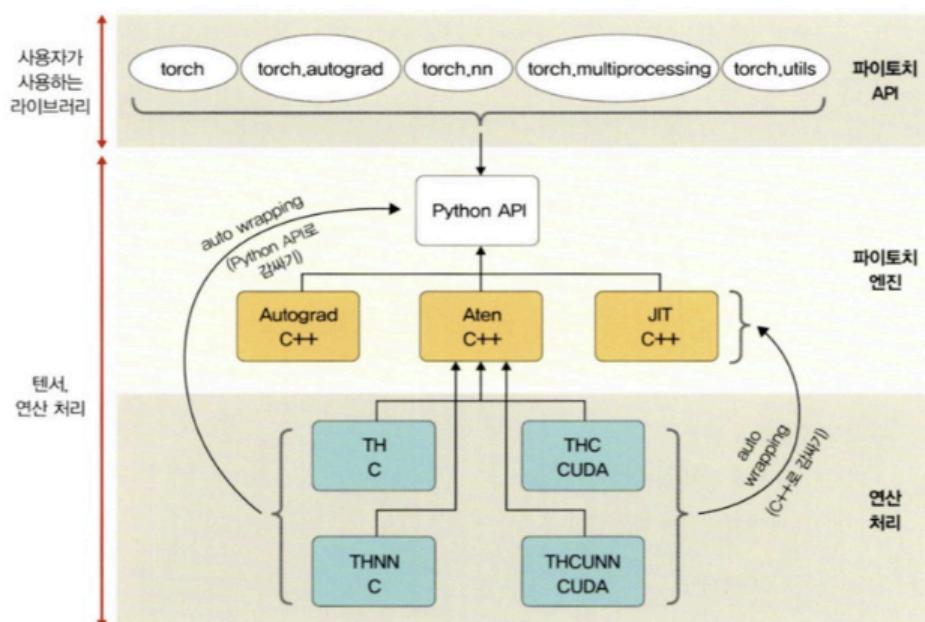
3. 파이토치 아키텍처

파이토치는 크게 3계층으로 나뉩니다.

1. API 계층: `torch`, `torch.nn`, `torch.autograd` 같은 고수준 패키지
2. 엔진 계층: Autograd C++, Aten C++, JIT C++ → 실제 연산 처리
3. 백엔드 계층: CPU/GPU 연산을 처리하는 C, CUDA 라이브러리

즉, 사용자는 Python API만 다루면 되고, 내부적으로는 C++/CUDA가 고속 연산을 수행한다.

▼ 그림 2-4 파이토치의 아키텍처



4. 파이토치 주요 패키지

- **torch**: 텐서 연산 기본 패키지
- **torch.autograd**: 자동 미분 패키지 (딥러닝 학습 핵심)
- **torch.nn**: 신경망 구축 모듈 (CNN, RNN 등)
- **torch.multiprocessing**: 병렬 연산 지원
- **torch.utils**: `DataLoader` 등 유ти리티

5. 텐서 저장 구조: 오프셋과 스트라이드

텐서는 단순 배열이 아니라 저장 방식(**Stride**, **Offset**)을 이해해야 한다.

- **Offset**: 시작 인덱스 위치
- **Stride**: 다음 요소로 이동하기 위해 건너뛰는 메모리 크기

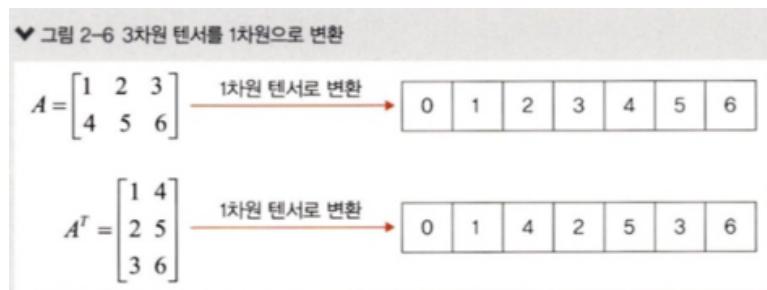
예: 2×3 텐서

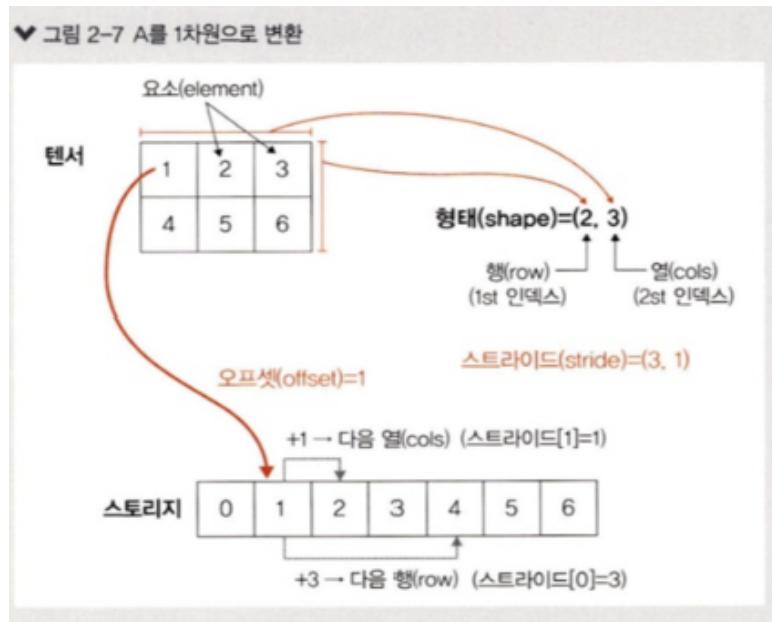
```
[[1, 2, 3],  
 [4, 5, 6]]
```

- **Shape**: $(2, 3)$
- **Stride**: $(3, 1) \rightarrow$ 한 행 넘어가려면 3칸 이동, 한 열 넘어가려면 1칸 이동

이 방식 덕분에 텐서의 부분 선택, 전치(transpose) 연산이 효율적으로 수행됨.

그림 2-6, 2-7에서는 **Offset**과 **Stride** 개념을 통해 텐서가 메모리에 어떻게 저장되는지를 설명다.





2.2 파이토치 기본 문법

1. 텐서 다루기

파이토치에서 가장 먼저 익혀야 할 문법은 [텐서 생성과 변환](#)이다.

```
import torch

# 텐서 생성
x = torch.tensor([[1, 2], [3, 4]])
print(x)

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

# 데이터 타입 지정
x_float = torch.tensor([[1, 2], [3, 4]]], dtype=torch.float64)

# 넘파이 변환
import numpy as np
x_np = x.numpy()
x_back = torch.from_numpy(x_np)
```

출력:

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

2. 데이터 준비

데이터는 Pandas, Numpy와 함께 불러오거나, PyTorch 내장 `Dataset` 과 `DataLoader` 를 활용한다.

(1) Pandas 활용

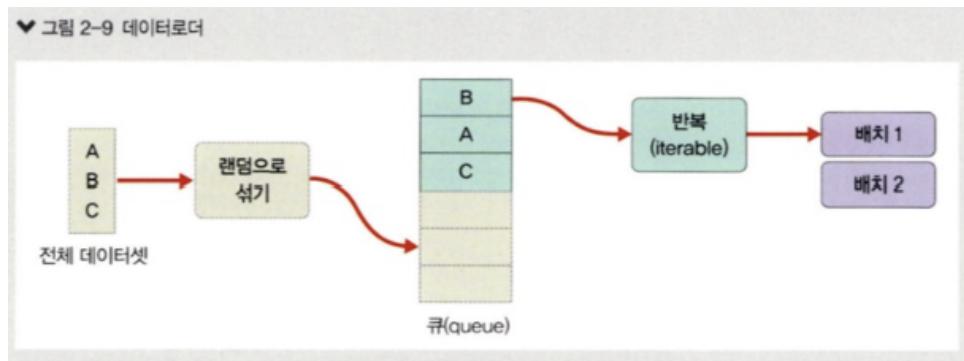
```
import pandas as pd  
import torch  
  
data = pd.read_csv('class2.csv')  
x = torch.from_numpy(data['x'].values).unsqueeze(dim=1).float()  
y = torch.from_numpy(data['y'].values).unsqueeze(dim=1).float()
```

(2) 커스텀 데이터셋 정의

`torch.utils.data.Dataset` 을 상속받아 `__len__` 과 `__getitem__` 을 구현.

```
from torch.utils.data import Dataset, DataLoader  
  
class CustomDataset(Dataset):  
    def __init__(self, csv_file):  
        self.label = pd.read_csv(csv_file)  
    def __len__(self):  
        return len(self.label)  
    def __getitem__(self, idx):  
        sample = torch.tensor(self.label.iloc[idx, 0:3]).int()  
        label = torch.tensor(self.label.iloc[idx, 3]).int()  
        return sample, label  
  
dataset = CustomDataset('covtype.csv')  
loader = DataLoader(dataset, batch_size=4, shuffle=True)
```

그림 2-9 데이터 로더 구조도는 전체 데이터셋을 배치 단위로 쪼개고 랜덤 셔플링 후 모델로 전달하는 과정을 시각적으로 보여다.



3. 모델 정의

PyTorch에서 모델은 **Layer(계층)**, **Module(모듈)**, **Model(최종 네트워크)** 단계로 구성된다. `nn.Module` 을 상속받거나 `nn.Sequential` 을 이용해 신경망을 정의한다.

- 계층(layer): 선형 계층(`Linear`), 합성곱 계층(`Conv2d`) 등.
- 모듈(module): 여러 계층을 모아 구성.
- 모델(model): 최종적인 신경망 전체 구조.

(1) 단순 모델

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

(2) Module 상속

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layer = nn.Linear(1, 1)
        self.activation = nn.Sigmoid()
```

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

(3) Sequential 방식

```
model = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=5),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Linear(64*5*5, 10),
    nn.ReLU()
)
```

4. 모델 파라미터 설정

- 활성화 함수: ReLU, Softmax, Sigmoid 등 모델 내에서 지정 가능.
- 파라미터 설정
 - 손실 함수
 - `BCELoss` : 이진 분류
 - `CrossEntropyLoss` : 다중 클래스 분류
 - `MSELoss` : 회귀
 - Optimizer (최적화 알고리즘)
 - `torch.optim` 모듈에서 제공 (SGD, Adam, RMSprop 등)
 - `optimizer.step()` 으로 파라미터 갱신
 - `zero_grad()` 로 기울기 초기화

5. 모델 훈련

(1) 학습률 스케줄러

- 학습 과정 중 학습률을 동적으로 조절.
- 종류:

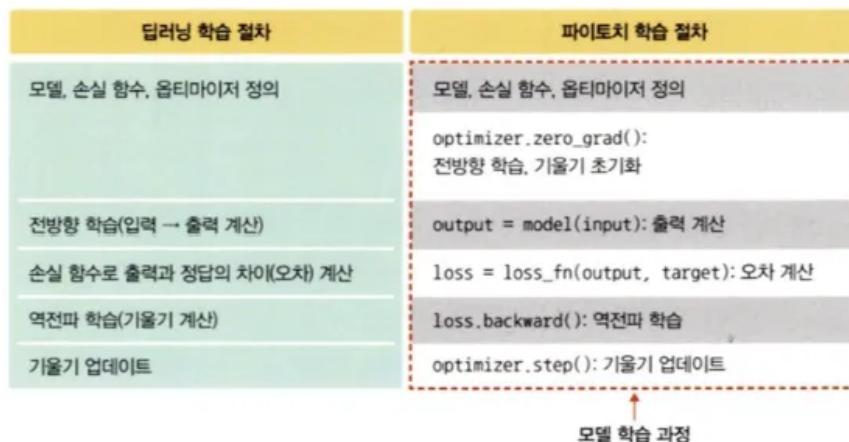
- StepLR : 일정 스텝마다 감소
- MultiStepLR : 여러 지점에서 감소
- ExponentialLR : 지수적으로 감소
- CosineAnnealingLR : 코사인 함수 기반 변동
- ReduceLROnPlateau : 성능 정체 시 감소

목적: 빠른 학습 초반 + 안정적인 수렴

(2) 학습 과정

1. `optimizer.zero_grad()` → 기울기 초기화
2. 모델 입력 → 출력 계산
3. 손실 함수(loss) 계산
4. `loss.backward()` → 역전파(기울기 계산)
5. `optimizer.step()` → 파라미터 갱신

→ 이 과정을 **epoch**(전체 데이터 1회 학습) 동안 반복.



6. 모델 평가

- 훈련/검증 데이터 분리

- `model.train()` : 학습 모드 (Dropout 활성화)
- `model.eval()` : 평가 모드 (Dropout 비활성화, `torch.no_grad()`로 불필요한 연산 방지)

- 평가 라이브러리
 - `torchmetrics`: 정확도, 혼동 행렬 등 제공
 - 예:

```
import torchmetrics  
metric = torchmetrics.Accuracy()
```

7. 학습 모니터링 (TensorBoard)

- 설치:

```
pip install tensorboard
```
- 사용 단계:
 1. `SummaryWriter` 객체 생성
 2. 학습 시 scalar 값 기록 (`writer.add_scalar`)
 3. `tensorboard --logdir=경로 --port=6006` 으로 실행
 4. 웹브라우저(`localhost:6006`)에서 시각화 확인

2.4 파이토치 코드 맛보기 (Car Evaluation 데이터셋)

1. 데이터 분포 확인

자동차 상태 예측(`car_evaluation.csv`) 데이터셋을 불러온 뒤, **출력(output)**의 분포를 확인한다.

- `unacc` (불가능) → 약 70%
- `acc` (허용) → 약 22%
- `good`, `vgood` → 약 7%

대부분의 데이터가 `unacc` 클래스에 몰려 있어, **불균형 데이터셋**임을 알 수 있다.

2. 범주형 데이터 전처리

파이토치 모델에 입력하기 위해서는 숫자 텐서로 변환해야 한다.

- `astype('category')`로 범주형 타입 변환
- `cat.codes`로 각 범주를 정수 인코딩

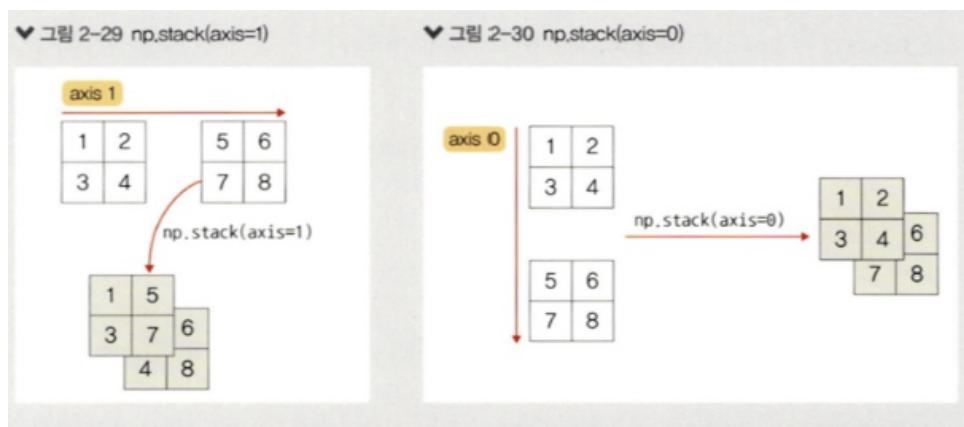
```
categorical_columns = ['price','maint','doors','persons','lug_capacity','safety']
for category in categorical_columns:
    dataset[category] = dataset[category].astype('category')
    dataset[category] = dataset[category].cat.codes
```

→ 이렇게 변환된 값은 NumPy 배열로 합쳐져 하나의 입력 텐서가 된다.

```
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

categorical_data = np.stack([price, maint, doors, persons, lug_capacity, safety], 1)
categorical_data[:10]
```

그림 2-27 ~ 2-30에서는 `np.stack` 과 `np.concatenate` 차이를 시각적으로 비교한다.



- `concatenate` : 기존 축을 따라 배열 연결
 - `stack` : 새로운 축을 추가하여 배열 결합
-

3. Tensor 변환

최종적으로 NumPy 배열을 PyTorch 텐서로 변환한다.

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

출력:

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

- `outputs` (라벨)는 `pd.get_dummies`를 통해 원-핫 인코딩 후 텐서로 변환.
-

4. 임베딩 크기 설정

범주형 데이터를 단순 숫자로 쓰지 않고, **임베딩 벡터**로 변환해 학습한다.

```
categorical_column_sizes = [len(dataset[col].cat.categories) for col 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)
```

출력:

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

→ 각 범주형 변수의 카테고리 개수와 임베딩 차원 쌍.

5. 데이터 분리

데이터셋을 훈련용 80%, 테스트용 20%로 나눈다.

```
total_records = 1728
test_records = int(total_records * .2)

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

출력 결과:

- 훈련 데이터: 1383개
- 테스트 데이터: 345개

6. 모델 정의

PyTorch의 `nn.Module` 을 상속받아 모델을 정의한다.

- 범주형 변수 → 임베딩(`nn.Embedding`)
- 은닉층(hidden layers) → `Linear → ReLU → BatchNorm → Dropout`
- 출력층 → `Linear`

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

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

```

여기서 중요한 부분:

- `self.all_embeddings`: 범주형 변수 각각을 임베딩
- `torch.cat`: 여러 임베딩을 하나로 합침
- **Dropout**: 과적합 방지
- **모델 객체 생성**

```

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

```

- **손실 함수 & 옵티마이저 정의**

```

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

```

- CPU/GPU 지정

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

- 모델 훈련(Train)

- `train_outputs` 와 `categorical_train_data`로 500 epoch 학습
- `optimizer.zero_grad() → loss.backward() → optimizer.step()`

```
# === 준비 ===
model.to(device)
categorical_train_data = categorical_train_data.to(device)
train_outputs = train_outputs.to(device=device, dtype=torch.long)

# === 훈련 ===
model.train() # 드롭아웃/배치정규화 '훈련 모드'
epochs = 500
aggregated_losses = []

for epoch in range(1, epochs + 1):
    # 순전파
    logits = model(categorical_train_data)      # [N, num_classes]
    loss = loss_function(logits, train_outputs) # train loss

    # 역전파
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # 로깅
    aggregated_losses.append(loss.item())
    if epoch % 25 == 1:
        print(f"epoch: {epoch:3d} loss: {loss.item():10.8f}")
```

```
print(f"epoch: {epoch:3d} loss: {loss.item():10.10f}")
```

7. 모델 테스트

```
# 테스트 데이터셋을 텐서로 변환
test_outputs = test_outputs.to(device=device, dtype=torch.int64)

# 모델 평가 모드 전환
with torch.no_grad():
    y_val = model(categorical_test_data.to(device))
    loss = loss_function(y_val, test_outputs) # 손실 계산
    print(f"Loss: {loss:.8f}") # ≈ 0.55
```

출력 결과:

```
Loss: 0.55124658
```

8. 평가 (Evaluation)

```
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# 로짓(logits) 값 출력 (softmax 이전 값)
print(y_val[:5])

# argmax로 클래스 예측
y_val = np.argmax(y_val.cpu().numpy(), axis=1)

# 혼동 행렬 & 성능 지표 출력
print(confusion_matrix(test_outputs.cpu().numpy(), y_val))
print(classification_report(test_outputs.cpu().numpy(), y_val))
print("Accuracy:", accuracy_score(test_outputs.cpu().numpy(), y_val))
```

출력 결과:

```
[[259 0]
 [ 85 1]]
```

	precision	recall	f1-score	support
0	0.75	1.00	0.86	259
1	1.00	0.01	0.02	86
accuracy		0.75	0.75	345
macro avg	0.88	0.51	0.44	345
weighted avg	0.81	0.75	0.65	345

0.7536231884057971

성능 지표 (Performance Metrics)

- 정확도 (Accuracy)

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

- 재현율 (Recall)

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

- 정밀도 (Precision)

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

- F1 점수 (F1-score)

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

9. 결과 정리

- 테스트 손실: 약 0.55
 - Accuracy: 약 75%
 - Precision / Recall / F1-score 함께 확인 가능
-

핵심 포인트

- 임베딩(Embedding): 범주형 데이터를 벡터로 변환
- 드롭아웃/배치정규화: 과적합 방지 & 학습 안정화
- GPU 자동 활용: `torch.cuda.is_available()`로 유연하게 처리
- 지표 활용: 단순 accuracy 외에 recall, precision, F1-score로 다양한 성능 평가
- 실제 학습 곡선: 손실이 점진적으로 감소하며 안정화됨을 확인