



# 1주차 발표

손소현 오수진 최하경

# 목차

---

#01 머신러닝

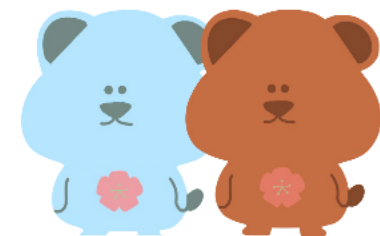
#02 딥러닝

#03 실습 환경 설정과 파이토치 기초

#04 코드 맛보기

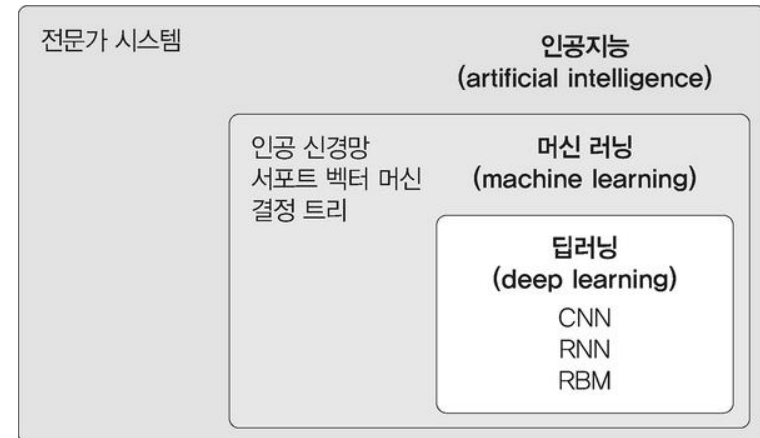


# 01. 머신러닝



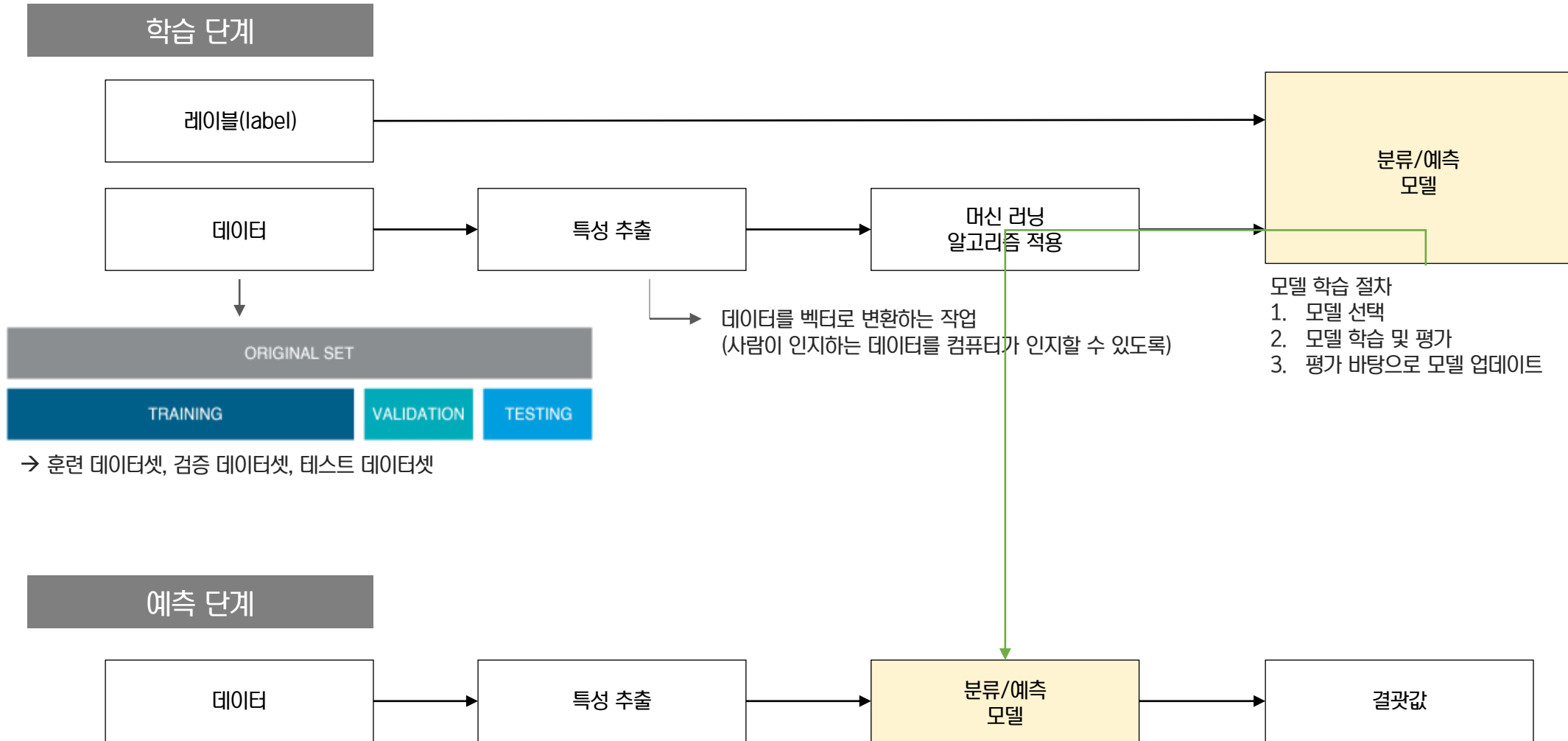
# #1.1 인공지능, 머신러닝, 딥러닝

인공지능	인간의 지능을 모방하여 사람이 하는 일을 컴퓨터가 할 수 있도록 하는 기술
머신러닝	각 데이터의 특성을 컴퓨터에 인식시키고 학습시켜 문제 해결
딥러닝	대량의 데이터를 신경망에 적용하면 컴퓨터가 스스로 분석한 후 답을 찾음



구분	머신러닝	딥러닝
동작원리	입력 데이터에 알고리즘을 적용하여 예측 수행	정보 전달하는 신경망을 사용하여 데이터의 특징 및 관계 해석
재사용	입력 데이터를 분석하기 위해 다양한 알고리즘 사용, 동일한 유형의 데이터 분석을 위한 재사용은 불가능	구현된 알고리즘은 동일한 유형의 데이터를 분석하는데 재사용
데이터	일반적으로 수천 개의 데이터 필요	수백만 개 이상의 데이터 필요
훈련시간	단시간	장시간
결과	일반적으로 점수 또는 분류 등 숫자 값	출력은 점수, 텍스트, 소리 등 어떤 것이든 가능

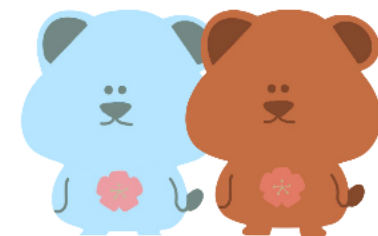
# #1.2 머신러닝



# #1.2 머신러닝

구분	유형	알고리즘
<b>지도 학습</b> :정답이 있는(레이블 된) 데이터를 활용해 데이터를 학습시키는 것	분류(classification)	<ul style="list-style-type: none"><li>- K-최근접 이웃(K-Nearest Neighbor, KNN)</li><li>- 서포트 벡터 머신(Support Vector Machine, SVM)<ul style="list-style-type: none"><li>- 결정 트리(decision tree)</li></ul></li><li>- 로지스틱 회귀(logistic regression)</li></ul>
	회귀(regression)	선형 회귀(linear regression)
<b>비지도 학습</b> :정답 레이블이 없는 데이터를 비슷한 특징끼리 군집화하여 새로운 데이터에 대한 결과를 예측하는 것	군집(clustering)	<ul style="list-style-type: none"><li>- K-평균 군집화(K-means clustering)</li><li>- 밀도 기반 군집 분석(DBSCAN)</li></ul>
	차원 축소(dimensionality reduction)	주성분 분석(Principal Component Analysis, PCA)
<b>강화 학습</b> :보상이 커지는 행동은 자주 하도록 하고, 줄어드는 행동은 덜 하도록 학습 진행	-	마르코프 결정 과정(Markov Decision Process, MDP)

## 02. 답러닝

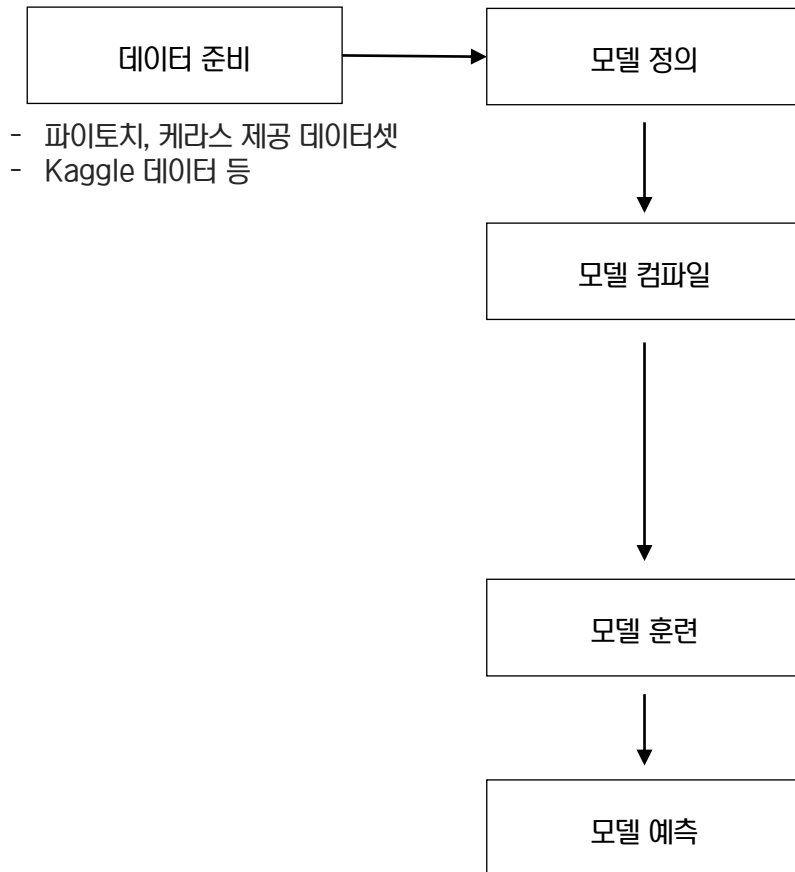


# #2.1 딥러닝

## 딥러닝:

인간의 신경망 원리를 모방한 심층 신경망 이론을 기반으로 고안된 머신 러닝 방법의 일종으로, 데이터를 컴퓨터가 처리 가능한 형태인 벡터나 행렬, 그래프 등으로 표현하고 이를 학습하는 모델을 구축하는 연구

## 딥러닝 학습 과정



- 파이토치, 케라스 제공 데이터셋
- Kaggle 데이터 등

- 신경망 생성
- 은닉층 ↑ → 성능 ↑ but 과적합 위험 ↑

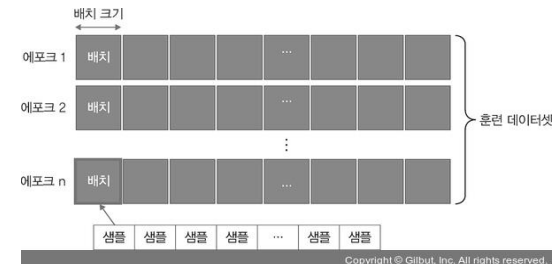
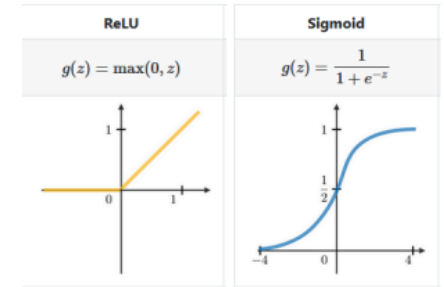
- 활성화 함수, 손실 함수, 옵티마이저 선택
  - 활성화 함수: 입력신호를 받아 출력 신호를 생성해주는 함수(Sigmoid, ReLU, Tanh 등)
  - 손실 함수: 회귀문제 - MSE, 분류문제 - Cross Entropy
  - 옵티마이저: 손실함수 기반으로 최적의 weight 구함(손실함수 최소화하는 w 계산, 아담(Adam) 등)
  - \*\*Cross Entropy → 실제값과 예측값이 맞는 경우 0으로 수렴하고, 틀린 경우 값이 커지기 때문에 실제값과 예측값의 차이를 줄이기 위한 방식

$$-\frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C L_{ic} \log(P_{ic})$$

$n$  = 데이터 개수,  $C$  = 범주 개수  
 $L$  = 실제 값,  $P$  = 실제 값에 대한 확률값

- 한번에 처리할 데이터양 지정
- 배치와 에포크 선택  
(배치: 한번에 몇 개의 샘플을 학습할 것인지 결정, 에포크: 훈련 횟수)
- 최적의 파라미터 및 하이퍼파라미터 값 확인

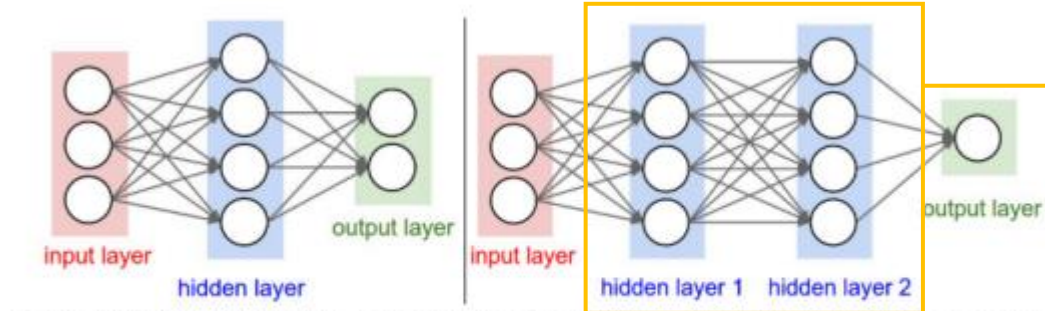
- 검증 데이터셋을 생성한 모델에 적용하여 실제 예측 진행
- 결과에 따라 파라미터 튜닝 또는 모델 재설계



Copyright © Gilbut, Inc. All rights reserved.

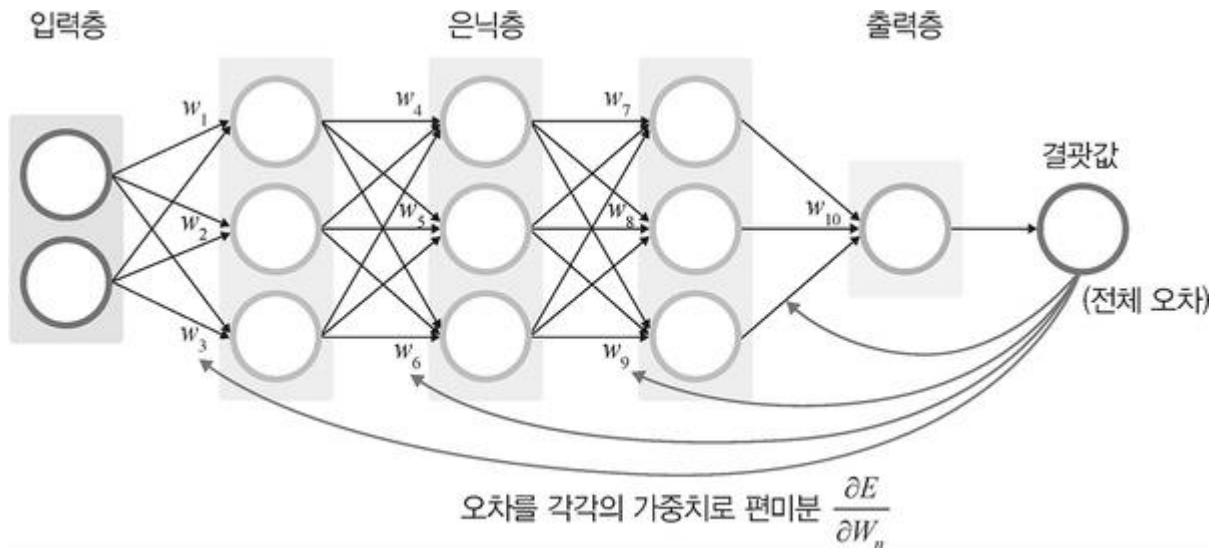


# #2.1 딥러닝



## 심층 신경망

데이터셋의 어떤 특성들이 중요한지 스스로에게 가르쳐줄 수 있는 기능 있음



## - 순전파:

네트워크의 입력층부터 출력층까지 순서대로 변수들을 계산하고 저장하는 것

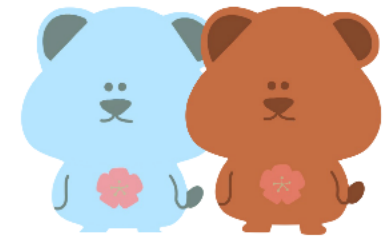
## - 역전파:

입력층의 입력 데이터에 대해 전방향 연산을 통해 계산된 오차를 네트워크에 존재하는 각각의 노드(퍼셉트론)에 역으로 전파하는 과정

## #2.2 딥러닝 학습 알고리즘

구분	유형	알고리즘
지도 학습	이미지 분류	<ul style="list-style-type: none"><li>- CNN(합성곱 신경망) → 이미지 분류, 이미지 인식, 이미지 분할<ul style="list-style-type: none"><li>- AlexNet</li><li>- ResNet</li></ul></li></ul>
	시계열 데이터 분석	<ul style="list-style-type: none"><li>- RNN(순환 신경망) → 역전파 과정에서 기울기 소멸 문제 발생<ul style="list-style-type: none"><li>- LSTM</li></ul></li></ul>
비지도 학습	워드 임베딩(word embedding) (단어를 벡터로 표현 - 자연어를 컴퓨터가 이해하도록 변환)	<ul style="list-style-type: none"><li>- Word2Vec</li><li>- GloVe</li></ul>
	군집(clustering)	<ul style="list-style-type: none"><li>- 가우시안 혼합 모델(GMM)</li><li>- 자기 조직화 지도(SOM)</li></ul>
	차원 축소(dimensionality reduction)	<ul style="list-style-type: none"><li>- 오토인코더(AutoEncoder)</li><li>- 주성분 분석(PCA)</li></ul>
전이 학습 : 사전에 학습이 완료된 모델을 이용해 원하는 학습에 미세 조정 기법을 이용하여 학습하는 방법	전이 학습	<ul style="list-style-type: none"><li>- 버트(BERT)</li><li>- MobileNetV2</li></ul>
강화 학습	-	마르코프 결정 과정(MDP)

### 03. 실습환경 설정과 파이토치 기초



# #3.1 실습 환경 설정과 파이토치 기초

- 파이토치 개요 : 특징 및 장점, 그리고 패키지 간단하게 정리해 설명
- 파이토치 특징 및 장점 : GPU에서 텐서 조작 및 동적 신경망 구축이 가능한 프레임 워크
- GPU : 연산속도를 빠르게 함. 병렬 연산에서 GPU의 속도는 CPU보다 훨씬 빠르므로 딥러닝 학습에서 GPU는 필수!
- 텐서 : 파이토치의 데이터 형태, 단일 데이터 형식으로 된 자료들의 다차원 행렬. 간단한 명령어를 추가함으로써 연산을 수행할 수 있게
- 동적 신경망 : 훈련을 반복할 때마다 네트워크 변경이 가능한 신경망. 학습 중에 은닉층을 추가하거나 제거하는 등 모델의 네트워크 조작이 가능

-> 효율적인 계산, 낮은 CPU 활용, 직관적인 인터페이스라는 특징으로 요약 할 수 있다.

# #3.1 실습 환경 설정과 파이토치 기초

- 벡터, 행렬, 텐서 개념 설명

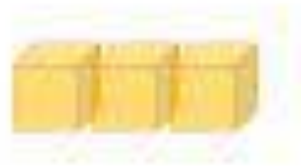
1차원 축(행) = axis 0 = 벡터

2차원 축(열) = axis 1 = 행렬

3차원 축(채널) = axis 2 = 텐서



Scalar  
rank=0



Vector  
rank=1



Matrix  
rank=2



3D Tensor  
rank=3

# #3.1 실습 환경 설정과 파이토치 기초

Scalar

1

Vector

$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$  or  $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

Matrix

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

Tensor

$\begin{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 3 & 4 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 \end{bmatrix} & \begin{bmatrix} 7 & 8 \end{bmatrix} \\ \begin{bmatrix} 9 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 2 \end{bmatrix} \end{bmatrix}$

# #3.1 실습 환경 설정과 파이토치 기초

오프셋(offset) : 텐서에서 첫 번째 요소가 스토리지에 저장된 인덱스

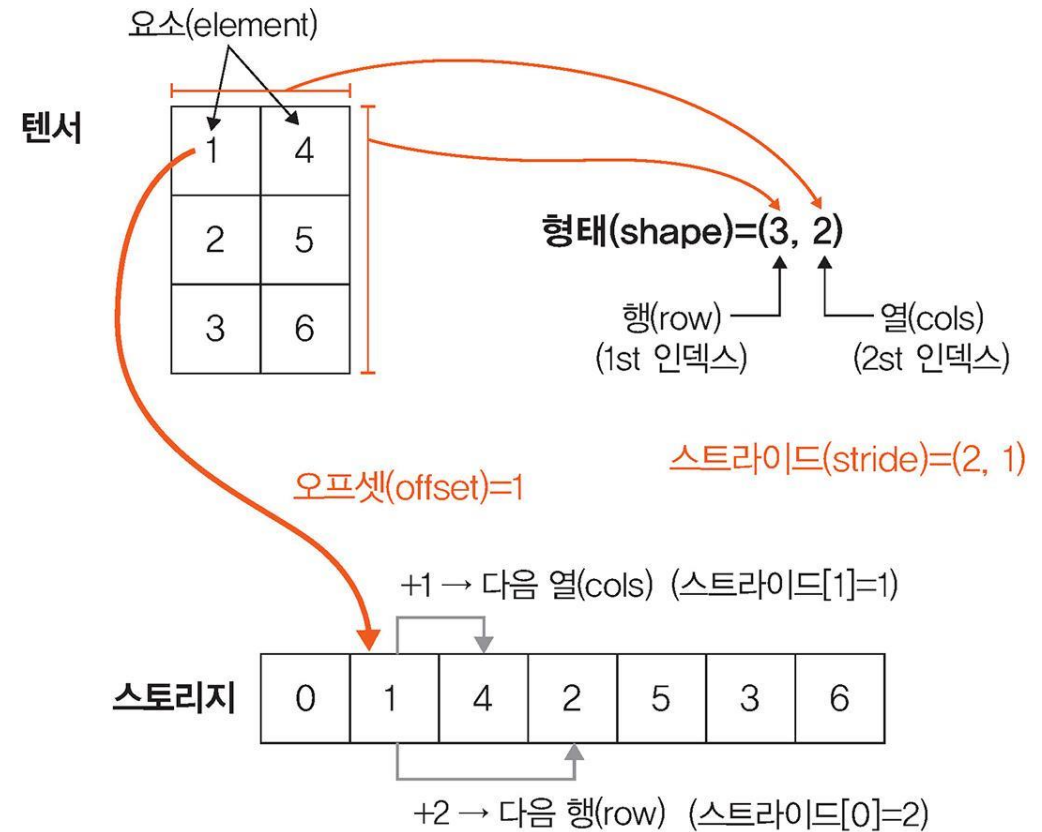
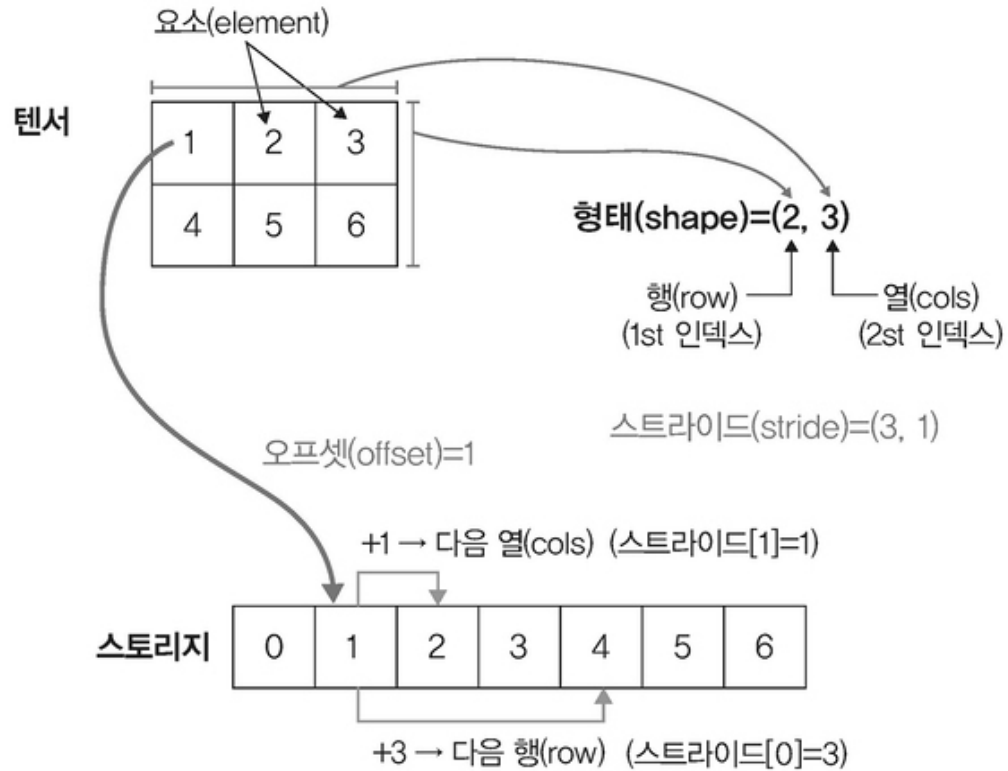
스트라이드(stride) : 각 차원에 따라 다음 요소를 얻기 위해 건너뛰기(skip)가 필요한 스토리지의

요소 개수 = 메모리에서의 텐서 레이아웃을 표현하는 것. 요소가 연속적으로 저장되기 때문에 행

중심으로 스트라이드는 항상 1. cnn에서도 나옴!

ex. ( 다음 행으로 가기 위해 몇 칸 가야 하는지, 다음 열로 가기 위해 몇 칸 가야하는지 )

# #3.1 실습 환경 설정과 파이토치 기초





# #3.1 실습 환경 설정과 파이토치 기초

- 텐서 생성 및 변환, 인덱스 조작, 자료형

## 텐서 생성

```
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
```

## 생성된 텐서의 모습

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

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

# #3.1 실습 환경 설정과 파이토치 기초

- 텐서 생성 및 변환, 인덱스 조작, 자료형

텐서 ndarray로 변환

```
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
```

# #3.1 실습 환경 설정과 파이토치 기초

- 텐서 생성 및 변환, 인덱스 조작, 자료형

## 텐서의 자료형

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

이외에도 다양한 유형의 텐서가 있음

# #3.1 실습 환경 설정과 파이토치 기초

## - 텐서 연산 및 차원 조작

넘파이의 ndarray처럼 다양한 수학 연산이 가능

텐서 간의 타입이 다르면 연산이 불가능.

예를 들어 FloatTensor와 DoubleTensor 간에 사칙 연산을 수행하면 오류가 발생

ex)

```
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
```

결과

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

# #3.1 실습 환경 설정과 파이토치 기초

텐서의 차원을 변경하는 가장 대표적인 방법은 **view**를 이용

이외에도 텐서를 결합하는 `stack`, `cat`과 차원을 교환하는 `t`, `transpose`도 사용.

`view`는 넘파이의 `reshape`과 유사하며 `cat`은 다른 길이의 텐서를 하나로 병합할 때 사용합니다.

또한, `transpose`는 행렬의 전치 외에도 차원의 순서를 변경할 때도 사용됩니다.

# #3.1 실습 환경 설정과 파이토치 기초

```
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))` ----- `-1`은 `(1, ?)`와 같은 의미로 다른 차원으로부터 해당 값을 유추하겠다는 것입니다. `temp`의 원소 개수( $2 \times 2 = 4$ )를 유지한 채 `(1, ?)`의 형태를 만족해야 하므로 `(1, 4)`가 됩니다.

```
print('-----')
```

`print(temp.view(-1, 1))` ----- 앞에서와 마찬가지로 `(?, 1)`의 의미로 `temp`의 원소 개수( $2 \times 2 = 4$ )를 유지한 채 `(?, 1)`의 형태를 만족해야 하므로 `(4, 1)`이 됩니다.

## #3.1 실습 환경 설정과 파이토치 기초

```
torch.Size([2, 2])
```

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

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

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

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

# #3.1 실습 환경 설정과 파이토치 기초

- 계층(layer): 모듈 또는 모듈을 구성하는 한 개의 계층으로 합성곱층(convolutional layer), 선형 계층(linear layer) 등
- 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



# #3.1 실습 환경 설정과 파이토치 기초

## 1. 단순 신경망을 정의하는 방법 :

- nn.Module을 상속받지 않는 매우 단순한 모델

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

# #3.1 실습 환경 설정과 파이토치 기초

2. 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

```
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
```

# #3.1 실습 환경 설정과 파이토치 기초

## 3. Sequential 신경망을 정의하는 방법 : 계산을

좀 더 **가독성이 뛰어나게** 코드로 작성.

- Sequential 객체는 그 안에 포함된 각 모듈을 순차적으로 실행

```
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))

        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)
        x = self.layer3(x)
        return x
```

model = MLP() ----- 모델에 대한 객체 생성

# #3.1 실습 환경 설정과 파이토치 기초

4. 함수로 신경망을 정의하는 방법 : Sequential을 이용하는 것과 동일하지만, 함수로 선언할 경우 변수에 저장해 놓은 계층들을 재사용할 수 있는 장점. 하지만 모델이 복잡해지는 단점이 있음.

```
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
```

## #3.2 파라미터들에 대한 설명

- 손실 함수(loss function)(목적 함수(objective function)):
  - 학습하는 동안 출력과 실제 값(정답) 사이의 오차를 측정.
  - 즉,  $w x + b$ 를 계산한 값과 실제 값인  $y$ 의 오차를 구해서 모델의 정확성을 측정
  - 훈련하는 동안 최소화될 값 주어진 문제에 대한 성공 지표

손실 함수	용도
이진 크로스 엔트로피(Binary Cross-entropy)	이진 분류
범주형 크로스 엔트로피(Categorical Cross-entropy)	다중 분류
평균 제곱 오차(Mean Squared Error)	회귀

## #3.2 파라미터들에 대한 설명

- 옵티마이저(optimizer):
  - 손실 함수를 기반으로 네트워크가 어떻게 업데이트될지 결정.
  - optimizer는 step() 메서드를 통해 전달받은 파라미터를 업데이트함
  - 모델의 파라미터별로 다른 기준(예 학습률)을 적용
- torch.optim.Optimizer(params, defaults)는 모든 옵티마이저의 기본이 되는 클래스.
- zero\_grad() 메서드는 옵티마이저에 사용된 파라미터들의 기울기(gradient)를 0으로
- torch.optim.lr\_scheduler는 에포크에 따라 학습률을 조절할 수 있음
  - › 자세한 내용은 '4장 딥러닝 시작' 에서 다시 다룹니다.

## #3.3 실습 환경 설정과 파이토치 기초

옵티마이저	특징
확률적 경사 하강법 (Stochastic Gradient Descent)	몇몇 데이터 샘플을 무작위로 추출하여 일부만 경사 하강법에 사용해 학습 속도 개선
RMSProp	기울기에 따라 학습률을 조절
Momentum	관성 개념을 추가
아담(Adam)	RMSProp과 Momentum의 아이디어를 합침

# #3.4 모델 훈련

## 딥러닝 학습 절차

모델, 손실 함수, 옵티마이저 정의

전방향 학습(입력 → 출력 계산)

손실 함수로 출력과 정답의 차이(오차) 계산

역전파 학습(기울기 계산)

기울기 업데이트

## 파이토치 학습 절차

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

↑  
모델 학습 과정

Copyright © Gilbut, Inc. All rights reserved.



## #3.5 모델 평가

모델 평가 : 주어진 테스트 데이터셋을 사용하여 모델을 평가합니다. 모델에 대한 평가는 함수와 모듈을 이용하는 두 가지 방법

함수)

```
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
```

## #3.5 모델 훈련

모듈)

```
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,))
```

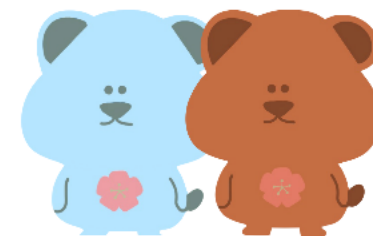
```
    acc = metric(preds, target)
```

```
    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)
```

## 04. 코드 맛보기



# #4.3 코드 맛보기

## #01 데이터 불러오기 + 확인

```
# 데이터 불러오기
data=pd.read_csv("./car_evaluation.csv")
data.head()
```

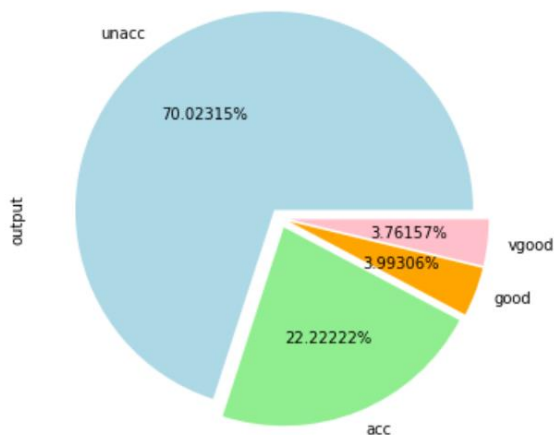
	price	maint	doors	persons	lug_capacity	safety	output
0	vhigh	vhigh	2	2	small	low	unacc
1	vhigh	vhigh	2	2	small	med	unacc
2	vhigh	vhigh	2	2	small	high	unacc
3	vhigh	vhigh	2	2	med	low	unacc
4	vhigh	vhigh	2	2	med	med	unacc

## #01 데이터 칼럼 설명

- Feature
  - Price : 자동차 가격
  - Maint : 자동차 유지 비용
  - Doors : 자동차 문 개수
  - Persons : 수용 인원
  - Lug\_capacity : 수하물 용량
  - Safety : 안전성
- Output
  - Unacc / acc / good / very good (vgood)

# #4.3 코드 맛보기

## #02 간단한 EDA



## #03 범주형 데이터 : dataset[category] -> numpy array -> Tensor

## 카테고리 칼럼 변환

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

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

```
price=data['price'].cat.codes.values
```

```
maint=data['maint'].cat.codes.values
```

```
doors=data['doors'].cat.codes.values
```

```
persons=data['persons'].cat.codes.values
```

```
lug_capacity=data['lug_capacity'].cat.codes.values
```

```
safety=data['safety'].cat.codes.values
```

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

```
categorical_data[:10]
```

이때 Series에 바로 codes, categories 매서드 적용하면 에러 나기 때문에 cat 매서드 사용

- Categorical 객체는 categories와 codes 속성을 가짐

```
data['price'].values.categories
```

```
Index(['high', 'low', 'med', 'vhigh'], dtype='object')
```

```
data['price'].values.codes
```

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

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)

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]])

# #4.3 코드 맛보기

## #04 output 데이터 텐서로 변환

```
## get_dummies를 이용해 output 칼럼 변환
outputs=pd.get_dummies(data.output)
outputs=outputs.values
outputs=torch.tensor(outputs).flatten() ## 1차원 텐서로 변환

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

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

## #05 워드 임베딩 크기 설정

```
## 워드 임베딩 크기 설정
categorical_column_sizes=[len(data[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) ## (범주형 칼럼의 고유값의 수, 차원 크기)

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

- 워드 임베딩
  - 유사한 단어끼리 유사하게 인코딩 되도록 표현
  - 높은 차원의 임베딩일수록 단어 간 세부적인 관계 파악 가능 -> 단일 숫자로 변환된 배열을 N차원으로 변경

# #4.3 코드 맛보기

## #06 모델 설정

## 모델 네트워크 생성

```
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=[]
```

```
        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
```


워드 임베딩 + Dropout 층

선형 계층 + Activation 함수 + 배치 정규화 + Dropout 층

학습 데이터를 입력 받아 연산 진행 (forward propagation)

# #4.3 코드 맛보기

## 🔑 워드 임베딩

- 워드 임베딩 (Word Embedding) 
  - 단어의 벡터화 : 희소표현 방식 (one-hot encoding) vs 밀집표현 방식 (word embedding)
  - 어떤 단어 → 단어에 부여된 고유한 정수값 → 임베딩 층 통과 → 밀집 벡터 (임베딩 벡터)
  - 두 가지 방법

- ① 임베딩 층을 만들어 처음부터 훈련 데이터를 이용해 임베딩 벡터를 학습
- ② pre-trained word embeddings를 가져와 사용

- 임베딩 층 == 룩업 테이블
  - 임베딩 벡터 : 인공 신경망의 가중치 학습과 같은 방법으로 훈련

Word → Integer → lookup Table → Embedding vector



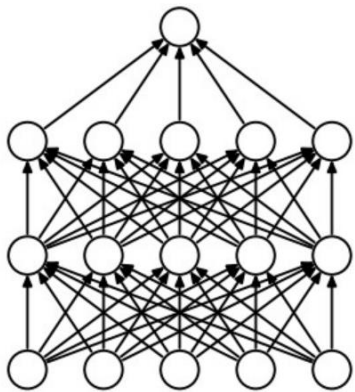
- nn.Embedding()의 파라미터
  - ① num\_embeddings : 임베딩할 단어의 개수 (단어 집합의 크기)
  - ② embedding\_dim : 임베딩할 벡터의 차원
  - ③ padding\_idx : 패딩을 위한 토큰의 인덱스 (선택적으로 사용)



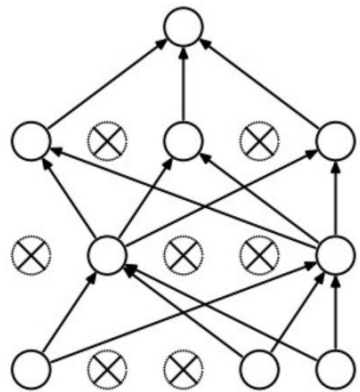
# #4.3 코드 맛보기

## 🔗 Dropout

- Dropout
  - 인공신경망 최적화 과정에서 overfitting을 방지해 모델의 정확도를 높여 줌 (정규화 방법 중 하나)
  - 훈련 과정에서 해당 층의 노드들을 랜덤으로 삭제해 훈련 데이터의 일부만 파라미터에 영향을 주도록 조정
  - 테스트 할 때는 모든 노드들을 사용함

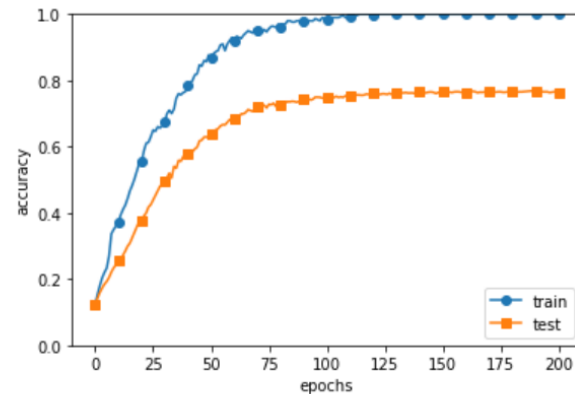


(a) Standard Neural Net

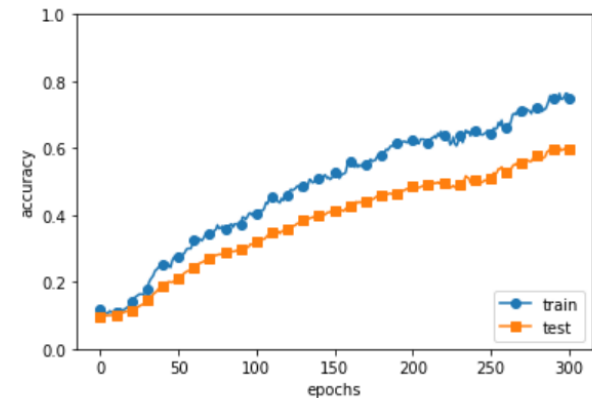


(b) After applying dropout.

드롭아웃 적용 전



드롭아웃 적용 후



# #4.3 코드 맛보기

## 📌 ReLU

- 활성화 함수 (Activation function)
  - 딥러닝 모델의 레이어 층을 깊게 가져가기 위해 노드에 입력된 값들을 비선형 함수에 통과시킴
  - 입력 데이터를 어떻게 다음 레이어로 출력하느냐를 결정하기 때문에 매우 중요
  - 종류 : 시그모이드, tanh, ReLU, leaky ReLU 등
  - gradient vanishing 문제를 해결하는 ReLU 함수를 가장 많이 사용

“

선형함수인  $h(x)=cx$ 를 활성화함수로 사용한 3층 네트워크를 떠올려 보세요.

이를 식으로 나타내면  $y(x)=h(h(h(x)))$ 가 됩니다. 이는 실은  $y(x)=ax$ 와 똑같은 식입니다.

$a=c^3$ 이라고만 하면 끝이죠. 즉, 은닉층이 없는 네트워크로 표현할 수 있습니다.

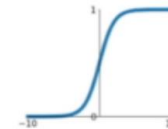
뉴럴네트워크에서 층을 쌓는 혜택을 얻고 싶다면 활성화함수로는 반드시 비선형 함수를 사용해야 합니다.

밑바닥부터 시작하는 딥러닝 中

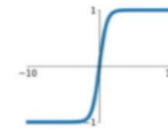
”

**Sigmoid**

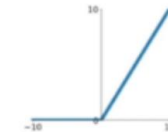
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



**tanh**  
 $\tanh(x)$

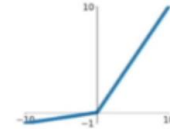


**ReLU**  
 $\max(0, x)$



**Leaky ReLU**

$$\max(0.1x, x)$$

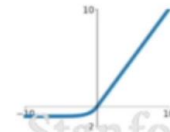


**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

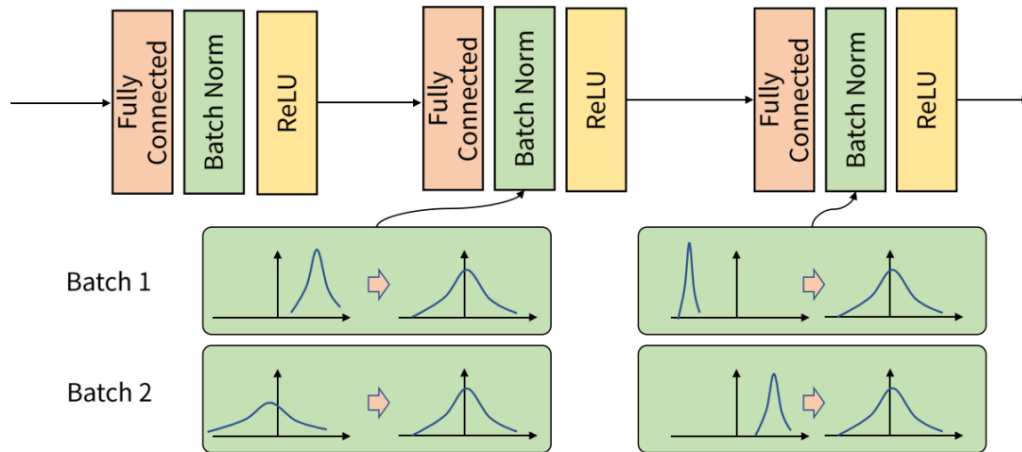
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



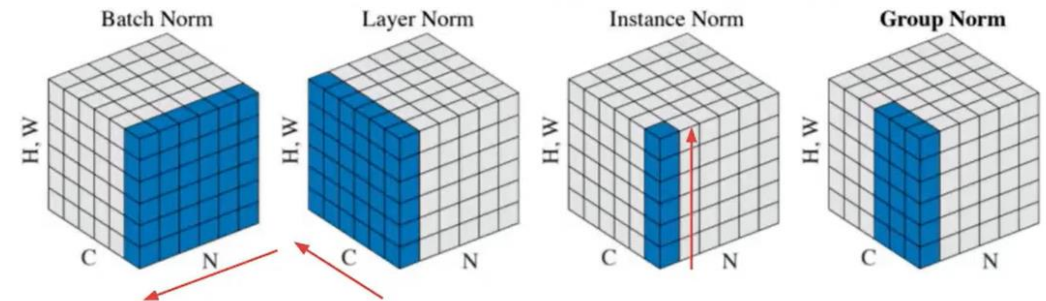
# #4.3 코드 맛보기

## 📌 배치 정규화 (Batch normalization)

- 배치 정규화
  - 대용량 데이터를 한번에 학습하지 못하기 때문에 batch 단위로 나눠서 학습
  - 이때 학습과정에서 계층별로 입력 데이터의 분포가 달라지는 현상 (internal covariant shift) 발생
  - 각 배치별로 gaussian 분포를 따르도록 학습 -> regularization 효과까지 있음
  - Train과 test data의 분포가 바뀌는 경우엔 정확한 예측이 불가능 -> Batch renormalization이란 방법 등장



- Layer Normalization (<https://arxiv.org/abs/1607.06450>)
- Instance Normalization (<https://arxiv.org/abs/1607.08022>)
- Group Normalization (<https://arxiv.org/abs/1803.08494>)



# #4.3 코드 맛보기

## #07 모델 초기화

## 모델 초기화

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

```
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)
  )
)
```

## #08 모델 파라미터 정의

## 모델 파라미터 정의

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

### Optimizer

- 인공지능망 최적화 방법
- 종류
  - SGD : saddle point, 작은 미니배치 등 문제점
  - SGD+Momentum / Adagrad : 뒤로 갈수록 적게 update
  - RMSProp
  - Adam : RMSProp+SGD+Momentum -> 가장 많이 쓰임
- First vs Second-order Optimization
  - First-order optimization
    - linear optimization
  - Second-order optimization
    - 이차 함수로 optimization 진행
    - convergence가 빠르다는 장점도 있지만, 헤시안 사이즈가  $O(n^2)$ 이고 역행렬은  $O(n^3)$ 이라 계산상 복잡하다는 단점이 있음
      - 오히려 linear이 더 빠르다는 의견도 있음

# #4.3 코드 맛보기

## #09 모델 학습

## 모델 학습

```
epochs=500
aggregated_losses=[]
train_outputs=train_outputs.to(device=device, dtype=torch.int64)
```

```
for i in range(epochs):
    i+=1
    y_pred=model(categorical_train_data)
    single_loss=loss_function(y_pred,train_outputs)
    aggregated_losses.append(single_loss)

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

```
optimizer.zero_grad()
single_loss.backward()
optimizer.step()
```

```
print(f'epoch:{i:3} loss:{single_loss.item():10.0f}')
```

Gradient update (Back propagation)

```
epoch:  1 loss:1.63960373
epoch: 26 loss:1.45137513
epoch: 51 loss:1.36734223
epoch: 76 loss:1.23154187
epoch:101 loss:1.07761014
epoch:126 loss:0.93326908
epoch:151 loss:0.82108617
epoch:176 loss:0.75516218
epoch:201 loss:0.69735128
epoch:226 loss:0.66804397
epoch:251 loss:0.63743061
epoch:276 loss:0.62235022
epoch:301 loss:0.60445195
epoch:326 loss:0.61003506
epoch:351 loss:0.59467822
epoch:376 loss:0.59520388
epoch:401 loss:0.59085774
epoch:426 loss:0.58432341
epoch:451 loss:0.57722622
epoch:476 loss:0.57491773
epoch:500 loss: 1
```

- optimizer.zero\_grad() : 모델 매개변수의 변화도 재설정
- loss.backward() : 예측 손실을 역전파 -> 각 매개변수에 대한 손실의 변화도 저장
- optimizer.step() : 역전파 단계에서 수집된 변화도로 매개변수 조정

# #4.3 코드 맛보기

## #08 모델 예측

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

Loss: 0.55256742

## #09 정확도 확인

```
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))
```

```
[[259  0]
 [ 84  2]]
```

	precision	recall	f1-score	support
0	0.76	1.00	0.86	259
1	1.00	0.02	0.05	86
accuracy			0.76	345
macro avg	0.88	0.51	0.45	345
weighted avg	0.82	0.76	0.66	345

0.7565217391304347

# THANK YOU

