

# 1주차

## ▼ 딥러닝의 3가지 큰 학습 틀

### 1 Supervised Learning (지도 학습)

| "정답이 있는 데이터를 통해 학습"

#### 특징:

- 각 입력에 대해 **\*\*정답(label)\*\***이 주어짐
- 모델은 예측을 하고, **실제 정답과 비교하며 학습**

#### 예시:

입력	정답
고양이 이미지	"cat"
이메일 본문	"스팸 여부"

#### 사용되는 분야:

- 이미지 분류 (e.g. 고양이 vs 개)
- 음성 인식
- 감성 분석

#### 대표 모델:

- CNN, RNN, Transformer + CrossEntropyLoss 등

### 2 Unsupervised Learning (비지도 학습)

| "정답 없이 구조나 패턴을 학습"

#### 특징:

- **label이 없음**
- 모델이 데이터 간의 **유사성, 군집, 분포** 등을 파악

## 예시:

- 문서 자동 군집화
- 이미지 압축
- 차원 축소 (PCA, t-SNE 등)

## 대표 기법:

- K-means
- Autoencoder
- GAN (생성적 적대 신경망) 일부 구조

---

## 3 Reinforcement Learning (강화 학습)

| "행동 → 보상 → 학습"

## 특징:

- 에이전트가 환경에서 **행동**을 하고
- 그에 따른 **\*\*보상(reward)\*\***을 받아
- **보상을 최대화하도록 전략(policy)**를 학습

## 예시:

- 알파고
- 자율 주행
- 게임 에이전트

## 용어:

개념	의미
상태 (state)	현재 환경 정보
행동 (action)	가능한 선택지
보상 (reward)	행동에 대한 결과
정책 (policy)	어떤 행동을 선택할지 정하는 함수

## ✅ 보너스: 최근 부각되는 하이브리드 학습 방식

이름	설명
<b>Self-Supervised Learning</b>	정답 없이도 일부 정보를 예측하게 해서 학습 (BERT 등)
<b>Semi-Supervised Learning</b>	소수의 라벨 + 다수의 무라벨 데이터를 활용

## 🎯 요약 표

구분	필요 데이터	목적	예시
지도 학습	입력 + 정답	정확한 예측	이미지 분류
비지도 학습	입력만	패턴 발견	군집화, 압축
강화 학습	환경 + 보상	전략 최적화	게임, 로봇

Parameterized mapping from images to label scores

→ 파라미터는 결국 모델과 같다.

모델은 파라미터로 나타낼 수 있음.

모델에 input을 넣어서 결과를 얻어내는 과정을 parameterized mapping이라고 표현한 것임.

## 📖 Lecture 2 – Image Classification with Linear Classifiers

### 1 문제 정의: 이미지 분류란?

“주어진 이미지를 사전에 정의된 클래스 중 하나로 분류하는 문제”

예시: 고양이/개/비행기/자동차 등의 클래스 중 하나로 이미지를 분류해야 함.

- 입력: 이미지 (예:  $32 \times 32$  픽셀 RGB → 총  $32 \times 32 \times 3 = 3072$ 차원 벡터)
- 출력: 하나의 클래스 라벨

### 2 컴퓨터가 보는 이미지: 시맨틱 갭 (semantic gap)

- 이미지는 \*\*단순한 숫자 배열(텐서)\*\*로 주어짐

예: 800×600×3 정수형 배열 (0~255 범위의 RGB)

- 인간은 이를 쉽게 인식하지만, 컴퓨터는 **의미 없이 숫자만 본다**

### 3 이미지 분류의 도전 과제들

과제	설명
<b>Viewpoint variation</b>	카메라 위치만 바뀌어도 픽셀이 전혀 달라짐
<b>Illumination</b>	조명이 다르면 색상이 달라짐
<b>Background clutter</b>	배경이 복잡하면 인식이 어렵다
<b>Occlusion</b>	객체가 가려져 있을 수 있음
<b>Deformation</b>	모양이 변형되어 있음
<b>Intraclass variation</b>	같은 클래스라도 매우 다양한 모양이 존재함
<b>Context</b>	주변 맥락에 따라 인식이 영향을 받음

### 4 데이터 기반 접근법 (Data-driven approach)

기존에는 "rule-based"로 시도했지만:

1. 학습용 이미지와 정답 라벨을 수집
2. 학습 알고리즘으로 분류기 학습
3. 새 이미지에서 정확도 평가

### 5 첫 번째 시도: 최근접 이웃 (K-Nearest Neighbor, K-NN)

원리:

- 모든 훈련 이미지를 저장
- 테스트 이미지가 주어지면 → 가장 가까운 K개의 이미지와 비교
- 다수결 투표로 클래스 결정
  - K개의 이웃 중에서 **\*\*가장 많이 등장한 클래스(label)\*\***를 정답으로 선택하는 방식

```
from collections import Counter

def majority_vote(labels):
```

```
count = Counter(labels)
return count.most_common(1)[0][0] # 가장 많이 나온 label 반환
```

#### ■ 주의할 점

문제	설명
<b>동점 발생</b>	예: [고양이, 개, 고양이, 개] → 2:2 동점 → 랜덤 선택 or tie-breaker 필요
<b>K는 홀수로 설정</b>	동점 피하려면 보통 K=3, 5, 7 등 홀수 사용
<b>가중 다수결</b>	가까운 이웃에게 더 큰 가중치를 주는 방식도 있음

#### ■ 예시

예시 1: K = 3인 경우

K개의 이웃의 클래스:  
[고양이, 개, 고양이]

→ 고양이 2표, 개 1표 → 예측 결과: 고양이

예시 2: K = 5인 경우

이웃들의 클래스:  
[고양이, 고양이, 개, 개, 개]

→ 고양이 2표, 개 3표 → 예측 결과: 개

### 거리 척도:

- **L1 거리 (Manhattan distance):** 계단식 거리
- **L2 거리 (Euclidean distance):** 직선 거리

### 시간복잡도:

- 학습:  $O(1)$
- 테스트:  $O(N)$  (→ 큰 데이터셋에서는 비효율적)

### 개선:

- FAISS 등 빠른 최근접 탐색 기법 존재

## 6 하이퍼파라미터 튜닝

- 하이퍼파라미터란?
  - 하이퍼파라미터는 모델의 성능을 좌우하는 외부 설정값으로, 학습 전에 사용자가 직접 정하고, 여러 실험을 통해 최적값을 찾아야 한다.

하이퍼파라미터	설명
학습률 (learning rate)	얼마나 빨리 학습할지 결정
배치 크기 (batch size)	몇 개의 데이터씩 학습할지
에폭 수 (epochs)	전체 데이터를 몇 번 반복할지
은닉층 크기 (hidden size)	신경망 레이어의 뉴런 수
활성화 함수	ReLU, sigmoid, tanh 등 어떤 걸 쓸지
정규화 계수 (lambda)	과적합 방지를 위한 규제 정도

- **K 값**: 가장 가까운 몇 개의 이웃을 볼지?
- **거리 척도**: L1 vs L2?
- **교차검증(Cross-validation)**: 데이터셋을 나눠 가장 좋은 K 찾기
  - fold가 5이면 데이터셋을 5개로 나눠서 4개는 학습, 한 개는 검증 데이터로 사용하는 방법

방법	하이퍼파라미터 선택 기준	적합성
Idea #1	training set 성능	❌ 과적합 위험
Idea #2	test set 성능	❌ 일반화 안 됨
Idea #3	validation 성능	✅ 추천 방법
Idea #4	cross-validation 평균 성능	✅ 소규모 데이터에 적합

- 과적합
  - 과적합(overfitting)은 머신러닝과 딥러닝에서 모델이 학습 데이터를 너무 지나치게 잘 외워서, 새로운 데이터(테스트 데이터)에는 성능이 떨어지는 현상
- Validation 데이터는 학습은 하지 않지만, **하이퍼파라미터를 조정하거나 모델을 개선하기 위해 중간중간 사용하는 평가 데이터**
  - 비유

데이터	비유
train	교과서를 공부하는 시간
validation	모의고사 보면서 약한 부분 보완하기
test	수능 시험 당일

## ▼ 추가 개념 설명

### ◆ 1. Training Set

- 모델이 **직접 학습**하는 데이터
- 예: 손글씨 이미지를 보고 숫자를 맞추도록 가중치 조정

### ◆ 2. Validation Set

- 학습된 모델을 중간에 시험해보는 용도
- 성능을 보면서 하이퍼파라미터를 튜닝하기 위해 사용
- 중요한 점:

모델이 validation을 "보긴 하지만"

→ 이 데이터로 **파라미터는 직접 학습하지 않음**

```
# 예: KNN에서 best_k 찾기
for k in [1, 3, 5, 7]:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    val_acc = model.score(X_val, y_val) # 여기서 val 사용
```

### ◆ 3. Test Set

- 모델의 **최종 평가용**
- 절대 사전에 보면 안 됨!
- 논문, 보고서, 대회에서 "이 모델의 정확도는 xx%"라고 할 때 쓰는 게 test set 입니다

## 7 K-NN: Summary

- 간단하고 직관적

- 고차원 이미지는 비효율적 (픽셀 기반 비교는 부적절)
- 실제로는 딥러닝이 등장하면서 잘 사용되지 않음
- Distance metric and K are hyperparameters

## 8 선형 분류기 (Linear Classifier)

수식 형태:

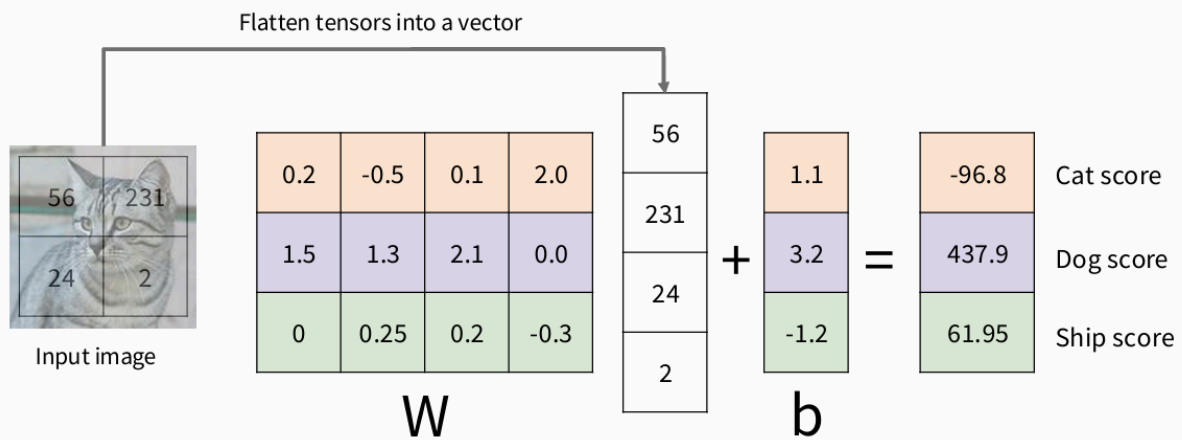
$$f(x, W) = Wx + b$$

- 행렬  $W$ 는 벡터  $x$ 를 새로운 공간으로 바꿔주는 변환 도구행렬
  - 새로운 관점으로 해석
  - 행렬 곱은 "렌즈" 같은 역할을 합니다.
    - 입력 벡터는 현실을 나타내는 신호
    - 행렬은 그 현실을 \*\*특정한 관점(기준, 방향)\*\*으로 바라보는 필터
    - 그 필터를 통과한 결과가 새로운 벡터  $y$
- 변수 설명 (괄호 안은 벡터, 행렬의 크기임)
  - $x$ : 벡터화된 이미지 ( $3072 \times 1$ ), 입력 이미지, 총 **3072개의 숫자** (픽셀 값)를 하나의 긴 벡터로 펼침
  - $W$ : 가중치 행렬 (클래스 수  $\times$  3072), 학습해야 할 가중치 (클래스별 중요도)
  - $b$ : 바이어스 (클래스 수), 각 클래스에 대해 따로 더해주는 보정값
  - 출력은 각 클래스에 대한 **score 벡터** (예: [cat: 3.2, dog: 5.1, car: -1.7])
- 최종 목표:
  - $W$ 와  $b$ 를 조정해서, 입력 이미지  $x$ 에 대해 올바른 class를 예측하도록 만드는 것
  - $W$ 와  $b$ 가 하이퍼파라미터

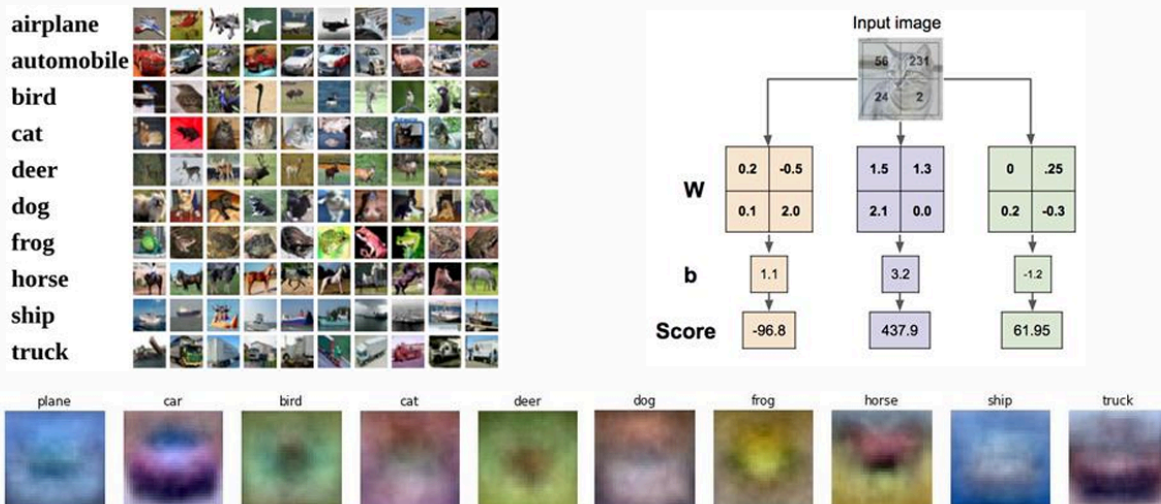
예시



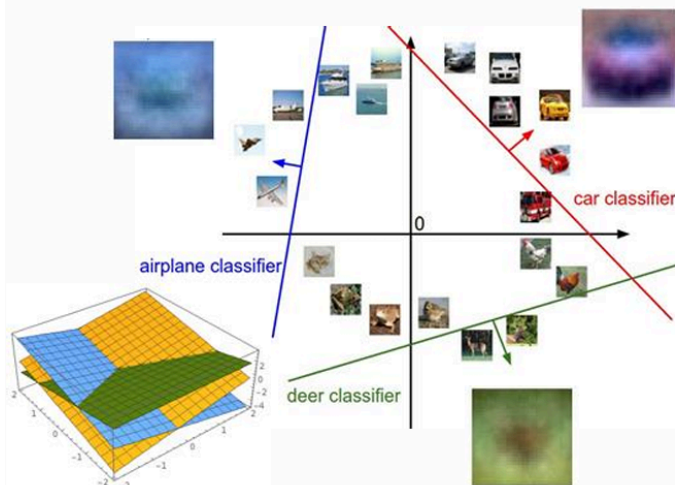
## Example with an image with 4 pixels, and 3 classes (cat/dog/ship) Algebraic Viewpoint



## Interpreting a Linear Classifier: Visual Viewpoint



## Interpreting a Linear Classifier: Geometric Viewpoint



$$f(x, W) = Wx + b$$



Array of 32x32x3 numbers  
(3072 numbers total)

Plot created using [Plotly Cloud](https://plotly.com/)

Cat image by [Nikita](https://www.shutterstock.com/image-photo/cat-image) is licensed under [CC-BY 2.0](https://creativecommons.org/licenses/by/2.0/)

- 각 데이터 라벨에 대한 일차함수를 우선 그려둔다. → 위 사진에서 빨간색, 초록색, 파란색
  - 입력값을 주어진 식으로 계산해서 점을 찍고, 그 점에서의 직선거리가 가장가까운 직선의 클래스로 라벨링 된다.

## 9 손실 함수 (Loss Function)

### Loss Function

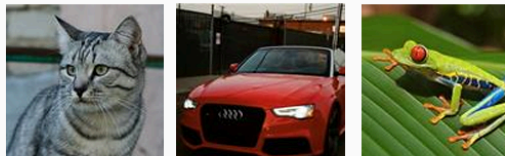
- 각 예제에 대해 "정답 클래스의 점수가 얼마나 낮은가"를 수치화
  - "모델이 얼마나 잘못됐는지"를 수치로 표현한 것이 손실 함수
- 모델의 예측이 얼마나 틀렸는지를 수치화함.

### Optimization

- 손실을 가장 작게 만들어주는  $W$ 를 찾는 것

### 예시:

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

A loss function tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

Where  $x_i$  is image and  $y_i$  is (integer) label

Loss over the dataset is a average of loss over examples:

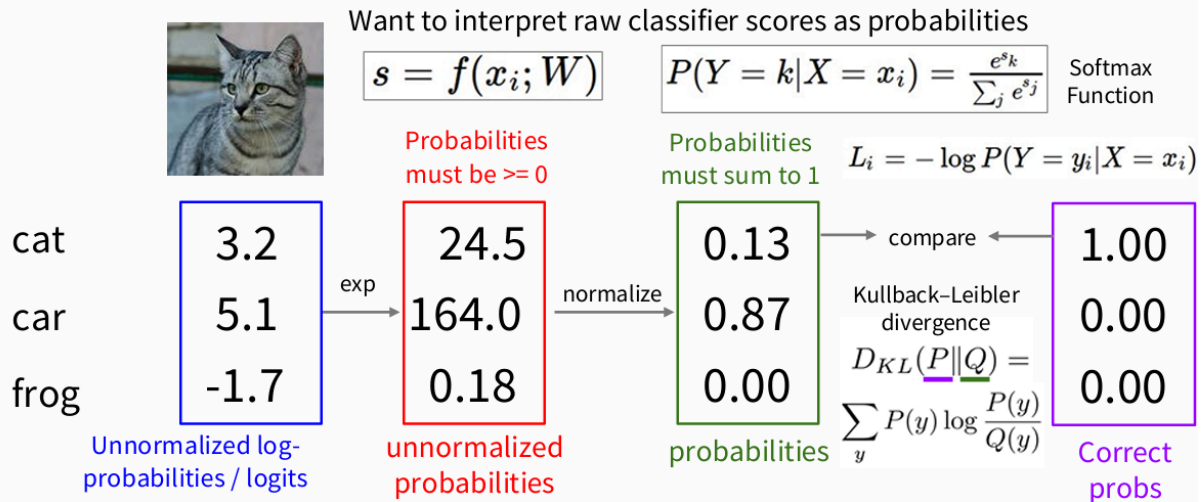
$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

- 정답이 'dog'인데 'cat' 점수가 더 높다면 → 큰 손실
- 전체 데이터셋에서 손실 평균을 취함:

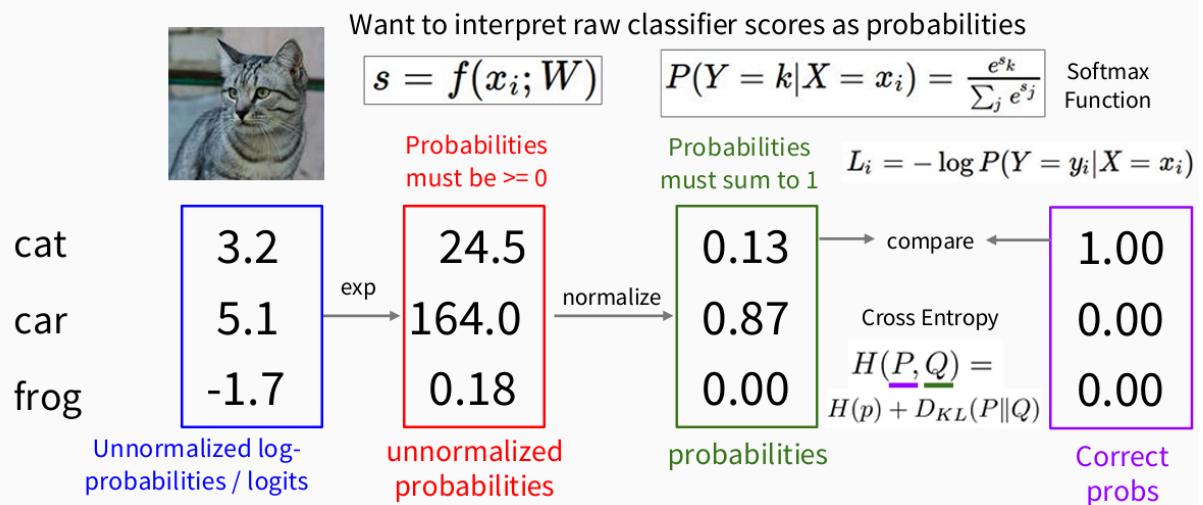
$$Loss = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

## 10 소프트맥스 분류기 (Softmax Classifier)

### Softmax Classifier (Multinomial Logistic Regression)



### Softmax Classifier (Multinomial Logistic Regression)



- 선형 분류기의 출력(score)를 **확률처럼 보이게** 만들고 싶음 → softmax 사용
  - 정답 벡터와 softmax 결과 벡터를 비교해서 얼마나 다른지 측정하는 개념이 **Kullback-Leibler Divergence**.
  - Cross Entropy** = 예측과 정답 확률 분포의 차이를 측정

Softmax 함수:

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

- process 과정
  - 지수화를 통해 양수화
    - 지수화 과정에서 numerical overflow 가능성이 있기 때문에 가장 높은 값을 구한 후 이 값으로 다른 모든 값을 빼는 방법도 있다.
  - 전체 확률의 합이 1이 되도록 정규화
  - softmax에서의 손실 계산

$$L_i = -\log P(Y = y_i | X = x_i)$$

## Kullback–Leibler Divergence (KL Divergence)

$$D_{KL}(P \| Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)}$$

"참된 분포 P"와 모델이 예측한 분포 Q 사이의 거리(차이)를 측정하는 비대칭적인 값

- P(y): 실제 정답 분포 (예: [1, 0, 0] for "cat")
- Q(y): 모델의 예측 확률 분포 (예: [0.13, 0.87, 0.00])
- 해석:
  - KL 값이 클수록 모델이 정답 분포에서 멀리 있음
  - KL은 항상 0 이상이며,  $D_{KL}(P \| Q) = 0$ 이면  $P = Q$

## Cross Entropy

$$H(P, Q) = - \sum_y P(y) \log Q(y)$$

혹은

$$H(P, Q) = H(P) + D_{KL}(P \| Q)$$

정답 분포 P를 기준으로, 모델의 분포 Q가 얼마나 잘 맞는지를 평가

→ 예측이 틀릴수록 **Cross Entropy** 값은 커짐

- 정답 확률 P(y)는 대부분 **one-hot 벡터** ([1, 0, 0])
- 이럴 때, H(P,Q)는 정답 클래스에 대한  $-\log Q(\text{정답})$ 와 같음

## 위 두 Loss Function의 관계

$$H(P, Q) = H(P) + D_{KL}(P \parallel Q)$$

- H(P): 고정된 값 (데이터 자체의 불확실성)
- 딥러닝 학습에서는 H(P)는 상수이므로 무시됨
- 결국, **Cross Entropy**를 최소화하는 것은 **KL Divergence**를 최소화하는 것과 같다
- 비교

항목	KL Divergence	Cross Entropy
수식	$\sum P(y) \log \frac{P(y)}{Q(y)}$	$-\sum P(y) \log Q(y)$
의미	두 분포 P, Q의 차이	예측 분포가 얼마나 정답 분포와 일치하는지
딥러닝에서	잘못 예측했을 때 벌점	loss function 자체
값이 0일 조건	P = Q	Q가 정답일 확률이 1일 때

## SVM vs Softmax 분류기: 개념적 차이

항목	SVM	Softmax
출력값	점수(score), 마진 기반	확률(probability), 정규화된 값
해석	해석 어려움 (비교만 가능)	클래스별 "신뢰도(confidence)"처럼 해석 가능

예: 입력 이미지에 대한 점수가 [12.5, 0.6, -23.0] 이라면

- SVM: 단순히 "고양이 > 개 > 배"일 뿐
- Softmax: [0.9, 0.09, 0.01] → 고양이일 확률이 90% 정도로 "매우 확신"

항목	SVM	Softmax
목표	정답 클래스 점수가 나머지보다 <b>margin만큼 크면 OK</b>	정답 클래스의 <b>확률을 최대화</b>

그 이후	마진 넘기면 더 이상 안 바꿈	확률 차이가 벌어질수록 계속 조정
학습 방식	"틀릴 위험이 높은 데이터"에 집중	전체 예측의 정밀도 계속 조정

## 실전에서의 성능은?

| 성능은 유사하지만, 문제 유형에 따라 유불리가 갈림

- **SVM 장점:**
  - 마진 기반 → 결정 경계가 뚜렷하게 필요한 경우 강력함
  - 잘 구분되는 클래스는 **무시하고**, 애매한 경계에 집중
- **Softmax 장점:**
  - 모델이 **확률적 예측**이 필요할 때 (예: 다중 선택, uncertainty 측정)
  - 미세한 클래스 간 차이를 계속 반영함

## ← END 마무리 요약

- 이미지 분류는 시맨틱 갭, 다양한 변형 등 많은 어려움 존재
- K-NN은 간단하지만 실전에서는 잘 안 씀
- **선형 분류기 + softmax + cross-entropy** 조합이 딥러닝의 핵심 구성 요소
- 손실 함수를 최소화하는  $W$ ,  $b$ 를 찾는 것이 학습의 핵심

## ▼ Python 문법 및 코드

### ▼ K = 1인 Nearest Neighbor classifier

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    # memorize training data
    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of si
        # the nearest neighbor classifier simply remembers all the training
```

```

self.Xtr = X
self.ytr = y

# Predict the label of the most similar training image
def predict(self, X):
    """ X is N x D where each row is an example we wish to predict lab
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype=self.ytr.dtype)

    # For each test image:
    # Find closest train image
    # Predict label of nearest image
    # loop over all test rows
    for i in range(num_test):
        # find the nearest training image to the i'th test image
        # using the L1 distance (sum of absolute value differences)
        distances = np.sum(np.abs(self.Xtr - X[i, :]), axis=1)
        min_index = np.argmin(distances) # get the index with smallest
        Ypred[i] = self.ytr[min_index] # predict the label of the nearest

    return Ypred

```

- 이 구현은 가장 기본적인 **1-NN 분류기**.
- `train()` 은 단순히 데이터를 저장하고, `predict()` 는 각 테스트 포인트마다 **가장 가까운 훈련 예제**를 찾아 **그 라벨을 그대로 복사**.
  - Train  $O(1)$ , predict  $O(N)$

## ▼ Class 문법

### 1. 클래스(Class)란?

클래스는 "설계도", 객체(Object)는 "설계도로 만든 실체"

예시로 생각:

- 자동차 공장의 "설계도" = 클래스

- 그 설계도로 만들어낸 실제 자동차 한 대 = 객체

## 기본 클래스 정의 문법

```
class 클래스이름:
    def __init__(self):
        초기화 코드

    def 메서드이름(self, 인자):
        동작 코드
```

## 2. 클래스의 실제 예제

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

    def bark(self):
        print(f"{self.name} says: 멍멍!")
```

- `class Dog` : Dog라는 클래스를 정의
- `__init__` : 객체가 생성될 때 실행됨
- `self.name = name` : 이름을 저장
- `bark()` : 짖는 동작

## 사용 방법

```
mydog = Dog("구름")
mydog.bark()
# 출력: 구름 says: 멍멍!
```

## 3. `self` 란?

- `self` 는 해당 객체 자신을 가리키는 변수



- 클래스 안에서 객체의 데이터를 저장하거나 접근할 때 **반드시** `self.속성명` 형식으로 씁니다.

예시:

```
class Test:
    def __init__(self, x):
        self.x = x # 객체 내부에 x 값을 저장

    def show(self):
        print(self.x)
```

## 4. 메서드(method)

클래스 안에 정의된 함수는 **메서드**라고 불러요.

다음 세 가지 유형이 있습니다:

종류	정의	특징
인스턴스 메서드	<code>def method(self)</code>	대부분 이걸 씀. 객체 내 데이터 사용
클래스 메서드	<code>@classmethod</code> + <code>def method(cls)</code>	클래스 전체에 적용되는 함수
정적 메서드	<code>@staticmethod</code> + <code>def method()</code>	<code>self</code> , <code>cls</code> 없이 일반 함수처럼 작동



## 5. 클래스 예제 정리

- 사람 클래스 예제

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"안녕하세요, 저는 {self.name}이고, {self.age}살입니다.")
```

```
p1 = Person("민수", 22)
p1.introduce()
```

```
# 출력: 안녕하세요, 저는 민수이고, 22살입니다.
```

## 6. 클래스 없이 짠 코드 vs 클래스 기반 코드 비교


- 클래스 없이:

```
def train(X, y):  
    return (X, y)  
  
def predict(X_train, y_train, X_test):  
    # 거리 계산, 예측 등...  
    return predictions
```

→ 데이터를 함수마다 다 전달해야 함

- 클래스 기반:

```
class KNN:  
    def train(self, X, y):  
        self.Xtr = X  
        self.ytr = y  
  
    def predict(self, X):  
        # self.Xtr, self.ytr 사용 가능!  
        return predictions
```

→ 한 객체( 클래스 인스턴스)에 모든 정보가 담겨 있어서 **코드 관리가 편리하고 구조화됨**

## 클래스는 언제 쓰나요?

- 복잡한 기능을 하나의 단위로 묶고 싶을 때
- 여러 개의 인스턴스가 필요할 때
- 상태(데이터)를 저장하면서 여러 동작을 정의하고 싶을 때

## 요약

요소	설명
----	----

<code>class</code>	클래스 정의 키워드
<code>__init__</code>	객체 생성 시 자동 호출되는 생성자
<code>self</code>	현재 객체 자신
메서드	클래스 내부 함수
속성 ( <code>self.x</code> )	객체가 가지고 있는 데이터

## ▼ Class와 C의 구조체

```
struct Point {
    int x;
    int y;
};
```

```
struct Point p1;
p1.x = 3;
p1.y = 4;
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
p1 = Point(3, 4)
```

- `struct` 는 데이터 박스
  - 오직 데이터를 묶는 용도
- `class` 는 데이터 + 행동을 가진 "생명체"
  - Python의 `class` 는 데이터 + 동작(함수)까지 포함

항목	C 구조체 ( <code>struct</code> )	Python 클래스 ( <code>class</code> )
데이터 저장	가능	가능
함수 포함	❌ (표준 C에선 안 됨)	✅ 메서드 정의 가능
캡슐화, 상속, 다형성	❌	✅ 객체지향 기능 전체 지원

생성자/소멸자	✗ 수동 초기화	✓ <code>__init__</code> , <code>__del__</code> 등
private/public 구분	없음	Python에서도 제한적이지만 <code>_name</code> 으로 표현 가능

## ▼ 확률론 복습

### 1. 사건의 확률

$$P(A) = \frac{\text{A가 일어나는 경우의 수}}{\text{전체 경우의 수}}$$

### 2. 조건부 확률

- 어떤 사건 A가 이미 일어났다고 "알고 있을 때", 사건 B가 일어날 확률을 구하는 것

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

- ML의 Softmax 분류기

$$P(Y = k \mid X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$