3강: 선형 분류 - SVM, 소프트맥스

개요

- k-최근접 이웃 분류기에는 몇 가지 단점이 있음
  - 모든 훈련 데이터를 저장해야하므로 공간 효율적이지 못함
  - 테스트 이미지 분류 시 모든 훈련 이미지와 비교해야하므로 계산 비용이 매우 큼
- 새로운 접근법
  - 데이터를 클래스 점수로 맵핑하는 점수 함수(score function)
  - 예측된 점수와 실제 정답 레이블간의 일치를 정량화하는 손실 함수(loss function)
  - 손실 함수를 최소화하는 점수 함수의 파라미터를 찾는 최적화 과정

점수 함수(score function)

- 이미지를 각 클래스에 대한 점수로 맵핑
- 점수는 특정 입력 데이터가 각 클래스에 속할 확률 또는 가능성을 나타내는 값
- 가장 높은 점수를 가진 클래스가 예측된 클래스가 됨

점수 함수

- N: 이미지 개수
  - ex) CIFAR-10 훈련 세트에서 N=50,000
- K: 클래스 개수
  - ex) CIFAR-10의 경우 K=10
- D: 각 이미지의 차원 크기
  - ex) CIFAR-10의 각 이미지는 32 x 32 x 3 = 3072 픽셀을 가지므로 D=3072

점수 함수

- 각각의 이미지 $x_i \in R^D$ 는 정답 레이블 y_i이 연결됨
  - i = 1, ..., N
  - yi = 1, ..., K
- 점수 함수는 각 이미지 $x_i \in R^D$ 를 R^K로 맵핑하는 함수

선형 분류기

- 선형 함수(linear function)은 가장 간단한 점수 함수
- 선형 분류기는 선형 함수를 점수 함수로 사용하는 분류기

$$f(x_i, W, b) = Wx_i + b$$

- x_i: D x 1 열벡터
- W: K x D 행렬
- b: K x 1 열벡터

선형 분류기

$$f(x_i, W, b) = Wx_i + b$$

- CIFAR-10의 경우
  - x_i: 3072 x 1 열벡터
  - W: 10x 3072 행렬
  - b: 10 x 1 열벡터
- 이미지(길이 3072 벡터)를 입력하면 10 x 1 열벡터가 출력됨
- 즉, 10개의 숫자(클래스 점수)가 출력됨

선형 분류기

$$f(x_i, W, b) = Wx_i + b$$

- 파라미터 W는 종종 가중치(weight)라고 불림
  - 가중치와 파라미터라는 용어를 흔히 바꿔 사용
- 파라미터 b는 편향 벡터(bias vector)라고 불림

주목할 사항 1

- 행렬 곱셈 Wx_i는 10개의 별도의 분류기를 병렬로 평가하는 것으로 볼 수 있음 (한 클래스에 하나씩)
- 각 분류기는 W의 각 행에 해당
  - W의 j번째 행을 w_j^T로 표현하면 Wx_i의 j번째 항은 $w_j^T x_i$
  - 이는 j번째 클래스에 대한 분류기로 생각할 수 있음

주목할 사항 2

- 입력 데이터 x_i, y_i는 고정되어 있지만, 파라미터 W, b는 조정할 수 있음
- 목표: 전체 훈련세트에 걸쳐 계산된 점수가 실제 정답 레이블과 일치하도록 파라미터 W, b 값을 설정하는 것
- 직관적으로, 올바른 클래스(정답 레이블)의 점수가 잘못된 클래스의 점수보다 높은 점수를 가지도록 함

주목할 사항 3

- 훈련세트가 파라미터 W, b를 학습하는데 사용되지만, 훈련이 완료되면 전체 훈련세트를 버리고 훈련된 파라미터만 유지할 수 있음
- 메모리 비용 측면에서 효율적

주목할 사항 4

- 테스트 이미지를 분류하는 것은 단일 행렬 곱셈 및 덧셈
- 이는 테스트 이미지를 모든 훈련 이미지와 비교(ex. 최근접 이웃 분류기)하는 것보다 훨씬 빠름

선형 분류기 해석

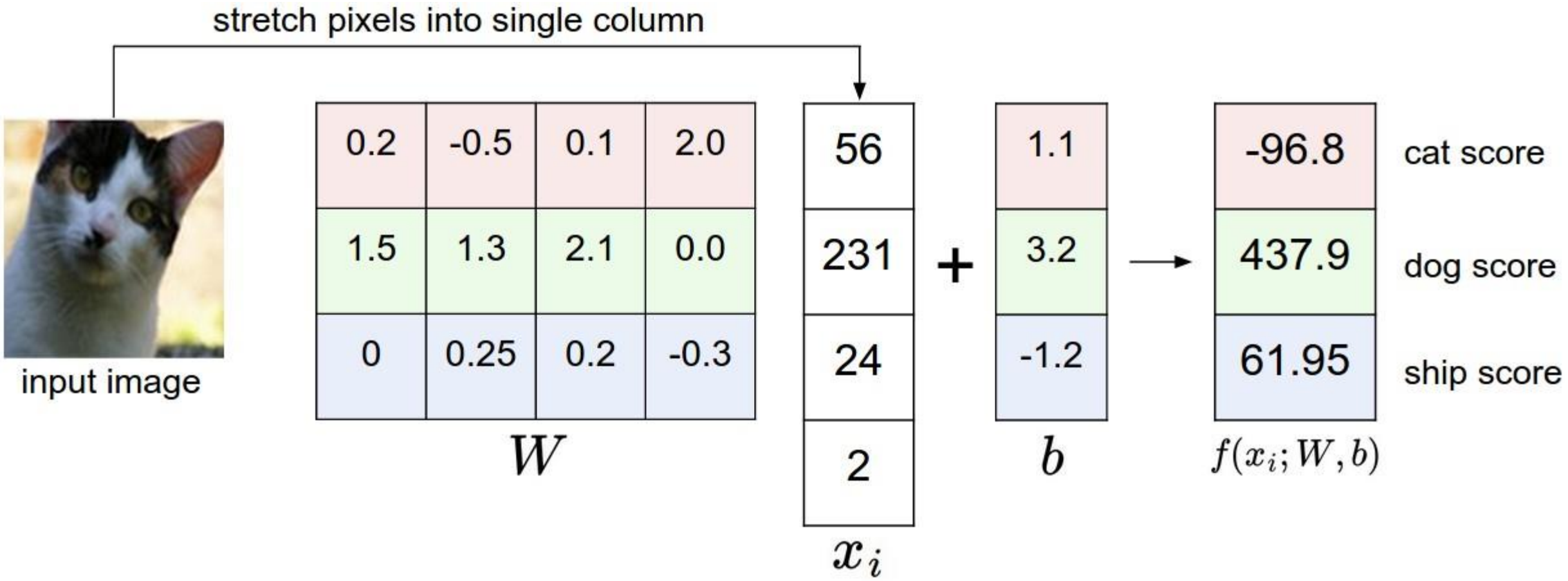
- 선형 분류기에서 클래스의 점수는 모든 픽셀 값(색상 채널 3개에 걸쳐)의 가중치 합으로 계산됨
- $w^T = \left[w^{(0)}, \cdots, w^{(3071)}\right]$
- $x^T = [x^{(0)}, \cdots, x^{(3071)}]$
- $w^T x = w^{(0)}x^{(0)} + w^{(1)}x^{(1)} + \cdots + w^{(3071)}x^{(3071)}$

선형 분류기 해석

- 가중치의 값에 따라 점수 함수는 이미지 특정 위치에서 특정 색상을 좋아하거나 싫어하게 됨
  - $w^T x = w^{(0)}x^{(0)} + w^{(1)}x^{(1)} + \cdots + w^{(3071)}x^{(3071)}$
- 예를 들어, "선박" 클래스는 많은 파란색이 있으면 더 가능성이 높아짐
- "선박" 분류기는 파랑 채널 가중치가 많이 양의 값을 가질 것으로 예상됨
- 반면, 빨강/초록 채널에서는 음의 가중치를 가질 것임

점수 계산 예시

- 이미지는 4개의 픽셀(단색)만 있다고 가정
- 클래스는 3개(고양이, 개, 선박)만 있다고 가정



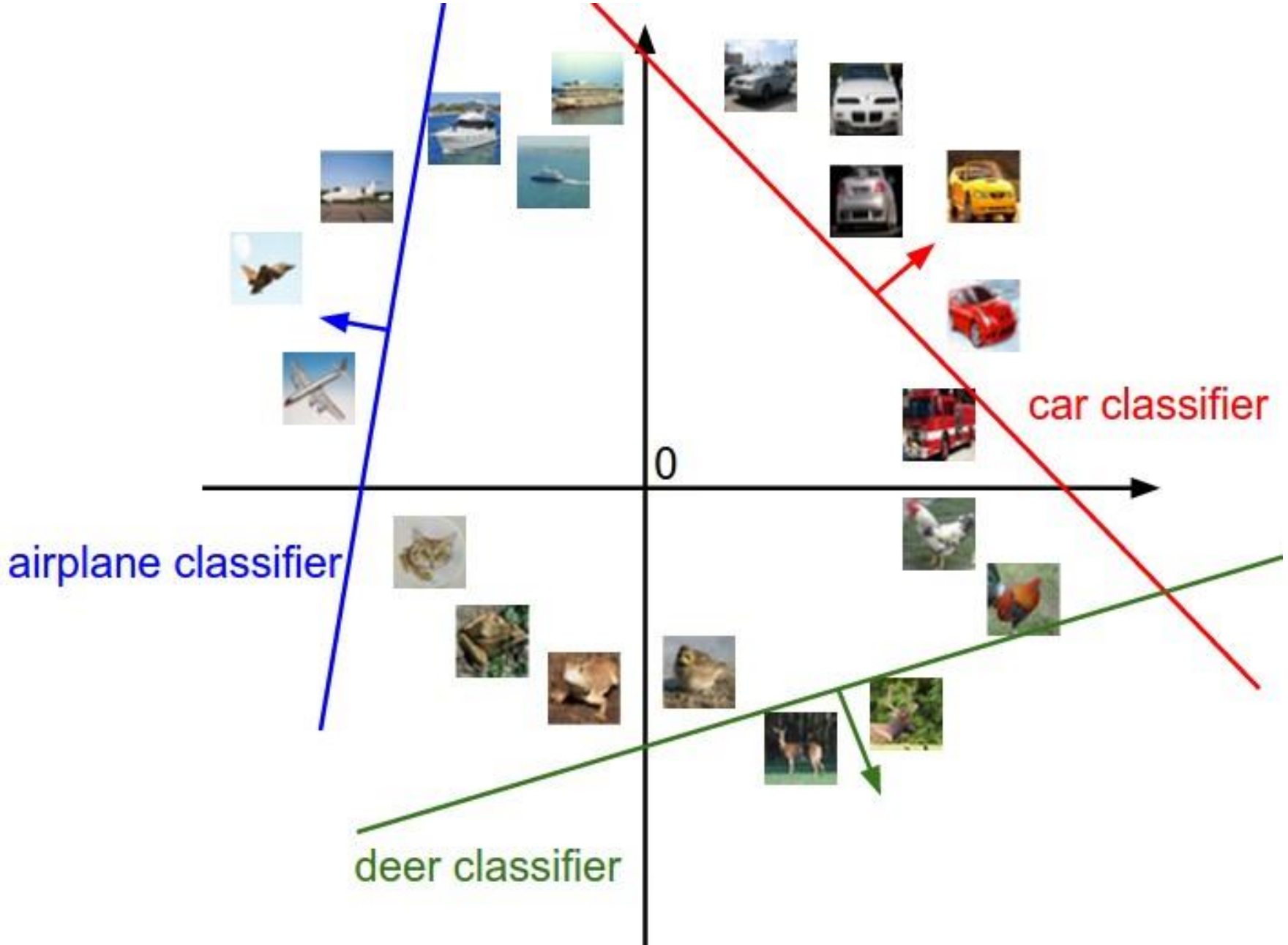
이미지를 고차원 점으로 해석

- 이미지를 고차원의 단일한 점으로 해석할 수 있음
- ex) CIFAR-10 이미지 32 x 32 x 3 픽셀은 3072 차원 공간($\mathbb{R}^{3072}$)에서의 한 점
- 전체 데이터셋은 (레이블이 붙은) 점들의 집합

이미지를 고차원 점으로 해석

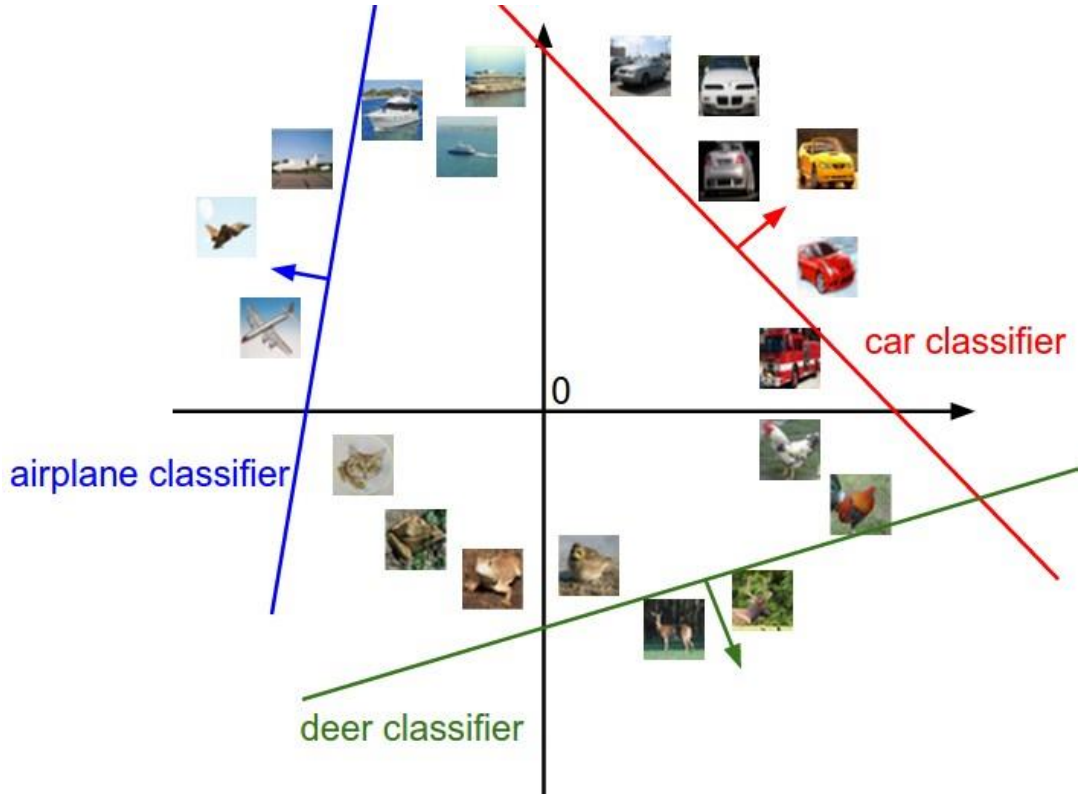
- 각 클래스의 점수는 이미지 벡터의 가중치 합이므로, 클래스 점수는 3072 차원 공간에서의 선형 함수
  - $w^T x = w^{(0)}x^{(0)} + w^{(1)}x^{(1)} + \cdots + w^{(3071)}x^{(3071)}$
- 3072차원 공간을 시각화할 수는 없지만, 2차원으로 압축한다면, 분류기 시각화 가능

이미지를 고차원 점으로 해석



이미지를 고차원 점으로 해석

- $y = Wx + b$
- W의 한 행 변경 시 분류기 선이 다른 방향으로 회전
- 편향 b 변경 시 분류기 선이 평행이동함

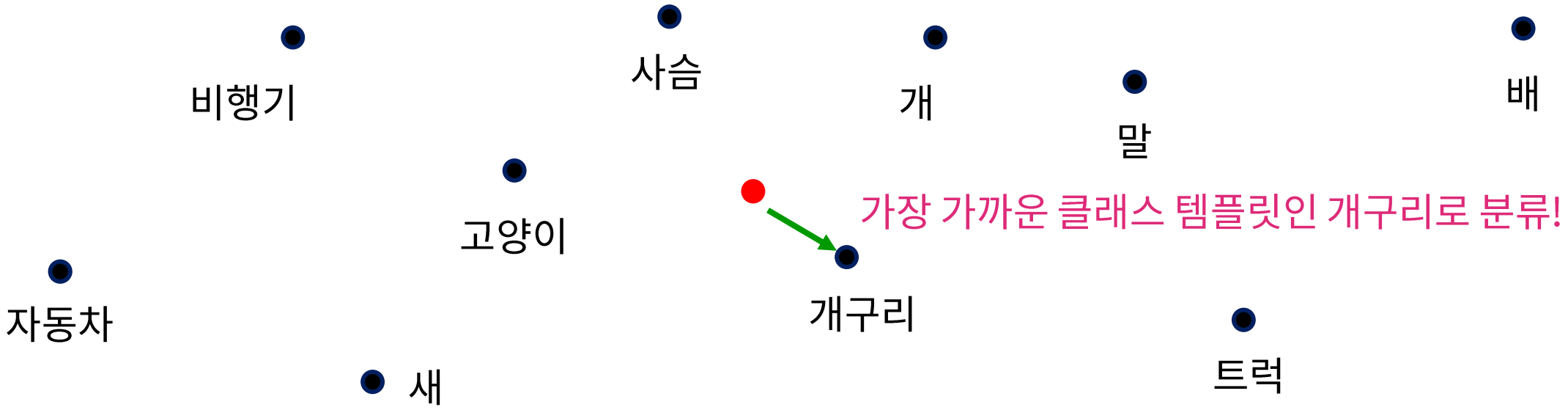


선형 분류기를 탬플릿 매칭으로 해석

- W의 각 행을 하나의 클래스에 대한 템플릿(혹은 프로토타입)으로 해석할 수 있음
- 이미지에 대한 각 클래스 점수는 각 템플릿을 이미지와 내적(dot product)하는 것이며, 각 템플릿과 얼마나 잘 맞는지에 대한 정도를 계산
- 선형 분류기는 주어진 이미지를 각 클래스의 템플릿과 비교하며, 가장 잘 맞는 템플릿(점수가 가장 높은 클래스)을 찾음
- 각 템플릿은 학습됨

선형 분류기를 탬플릿 매칭으로 해석

- 다른 생각 방식: 여전히 최근접 이웃 분류기를 수행하고 있지만, 수천개의 훈련세트 이미지 대신 각 클래스 당 하나의 이미지만 사용
- 이 하나의 이미지는 템플릿에 해당되며 학습됨
- 거리로는 L2 norm 대신 (음의) 내적을 사용



선형 분류기를 탬플릿 매칭으로 해석

- CIFAR-10의 예시 (학습된) 템플릿 가중치
- 예상대로, 배 템플릿에는 많은 파란색 픽셀이 포함됨
- 템플릿은 해당 클래스를 대표하는 평균 이미지 혹은 이상적(ideal) 이미지와 유사하게 생각할 수 있음
- 템플릿의 각 성분은 이미지의 해당 픽셀이 그 클래스에 얼마나 중요한지를 나타냄

선형 분류기를 탬플릿 매칭으로 해석

- 말 템플릿이 두 개의 머리를 가진 말을 포함하는 것처럼 보임
- 이는 데이터셋에 왼쪽, 오른쪽 향하는 말 두 가지가 있기 때문

선형 분류기를 템플릿 매칭으로 해석

- 자동차 분류기는 여러 모드를 하나의 템플릿으로 통합
- 모든 측면과 모든 색상의 자동차를 식별해야함
- 이 템플릿이 빨간색인 이유는 CIFAR-10 데이터세트에 빨간색 자동차가 많이 있다는 것을 암시

편향 트릭(bias trick)

- 두 파라미터 W와 b를 하나로 표현할 수 있음

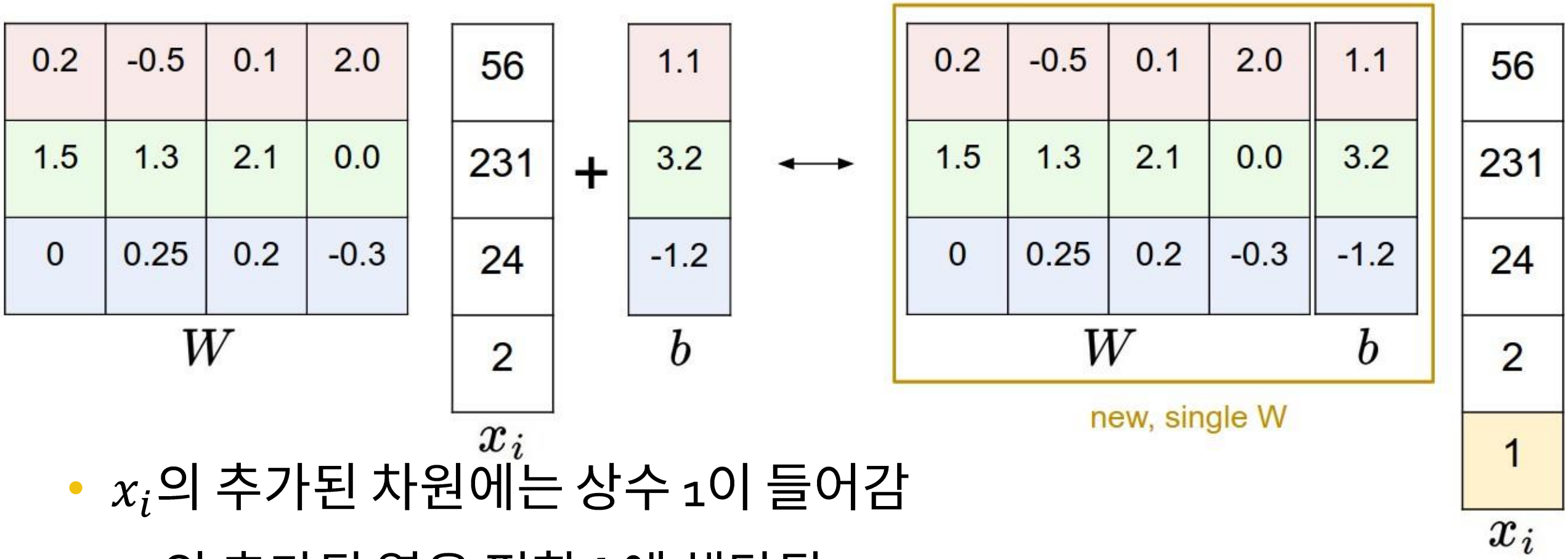
$$f(x_i, W, b) = Wx_i + b$$

- 벡터 x_i에 항상 상수 1을 가지는 추가 차원을 확장하여, 두 파라미터 W와 b를 모두 갖는 단일 행렬로 결합

$$f(x_i, W) = Wx_i$$

- x_i는 길이 3072 벡터에서 3073 벡터로 확장됨
- W는 10 x 3072 행렬에서 10 x 3073 행렬로 확장됨

편향 트릭(bias trick)



- x_i의 추가된 차원에는 상수 1이 들어감
- W의 추가된 열은 편향 b에 해당됨
- 따라서, 기존 선형 함수는 모든 입력 벡터에 상수 1 차원을 추가하고, 가중치 행렬을 1열 확장하는 것과 동일

전처리(preprocessing)

- 방금 예시에서는 원시 픽셀 값을 사용함 (0~255)
- 머신러닝에서는 입력 픽셀 값의 정규화(normalization)를 항상 수행하는 것이 매우 일반적인 관행
- 특히, 모든 입력 픽셀에서 평균을 빼서 데이터를 중심에 맞추는 것이 중요
  - 모든 픽셀이 대략 -127~127 값을 가지게 됨

전처리(preprocessing)

- 또다른 일반적인 전처리는 각 입력 이미지 픽셀을 스케일링하여 그 값이 [-1,1] 범위에 들도록 하는 것

배경지식 – 정규화(normalization)

- 정규화(normalization)란 머신러닝에서 다양한 범위와 단위를 가진 데이터를 모두 비슷한 기준으로 맞추는 과정
- 정규화의 중요성
  - 속도 향상: 학습 속도가 더 빠름. 모든 특징이 비슷한 범위를 갖게 되면, 학습 알고리즘이 최적의 해답을 찾기 위해 필요한 단계 수가 줄어듦
  - 정확도 향상: 일부 알고리즘은 특징의 범위나 크기에 민감함. 정규화를 통해 이런 민감성을 줄일 수 있음
  - 이해하기 쉬움: 데이터가 비슷한 기준으로 맞추어져 있을 때, 해당 데이터를 이해하거나 분석하기가 더 쉬움

배경지식 – 정규화(normalization)

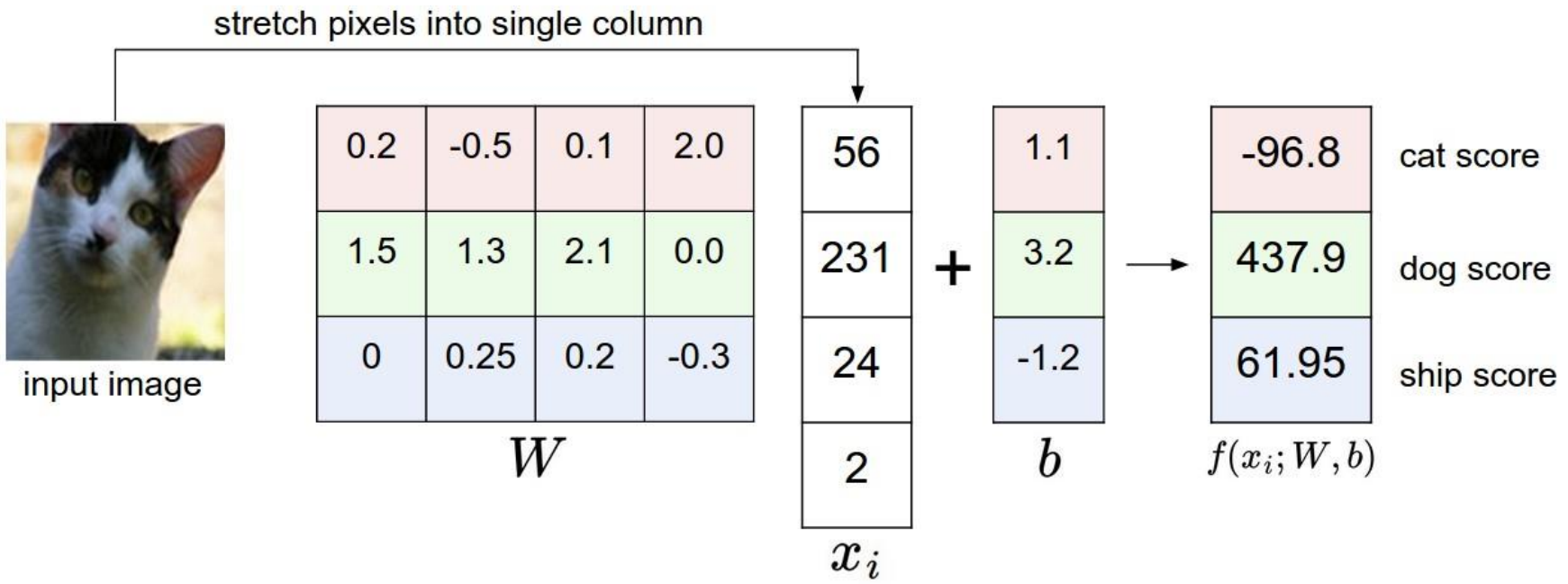
- Normalization과 regularization과의 차이?
  - 둘 다 한글로는 "정규화"
  - Normalization: 데이터를 특정 범위로 변환하는 과정
  - Regularization: 모델 파라미터 크기에 페널티를 주어 모델의 과적합을 방지하는 기법

손실 함수(loss function)

- 목표
  - 지금까지 점수 함수를 정의했고 이는 가중치 W로 파라미터화됨
  - 데이터 (x_i, y_i)는 고정되어있으며 주어짐
  - 목표: 가중치 W를 조정하여 훈련세트의 정답 레이블과 예측 레이블(클래스 점수)이 최대한 일치하도록 하는 것

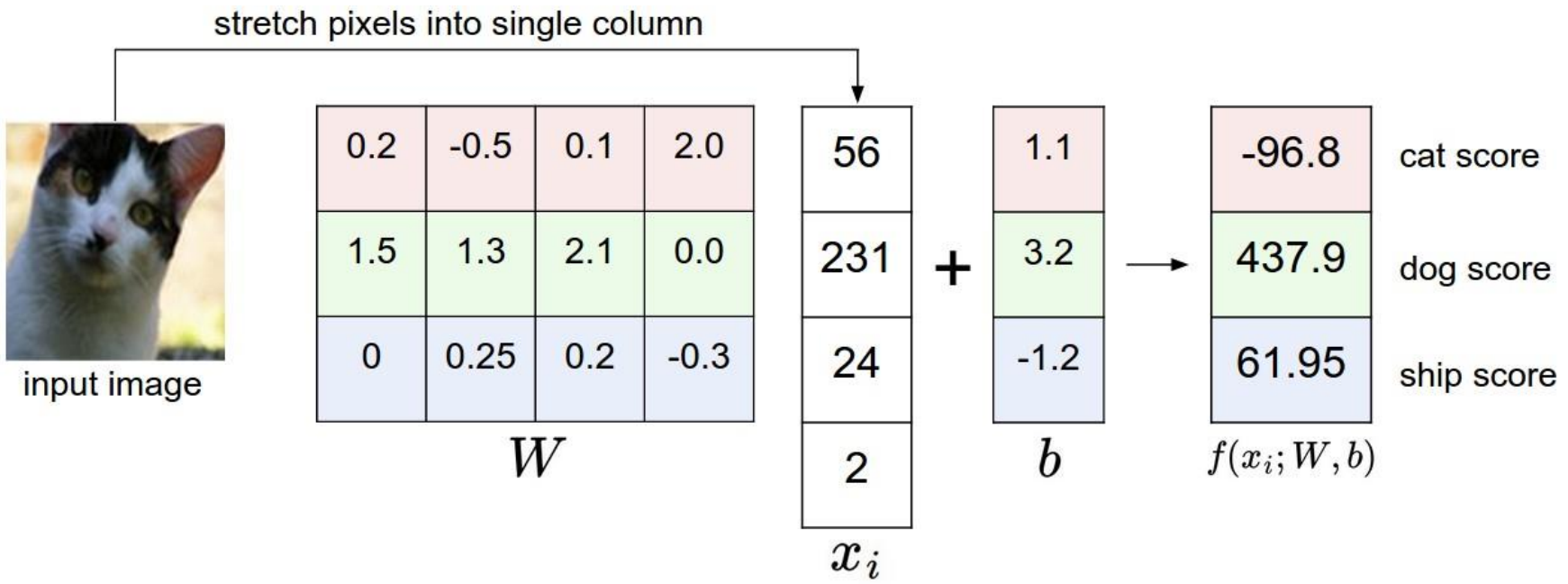
손실 함수(loss function)

- 아래 예시에서의 가중치 세트는 좋지 않았음
- 이 결과에 대한 불행함(불만족스러움)을 손실 함수(loss function)로 측정함
- 손실 함수는 종종 비용 함수(cost function) 혹은 목표(objective)로 불림



손실 함수(loss function)

- 직관적으로 분류를 잘 못하고 있으면 손실이 크게 되고, 분류를 잘 하고 있으면 손실이 작아짐



손실 함수 상세

- 손실 함수는 먼저 각 훈련세트 이미지에 대해 계산됨
- 입력 이미지에 대한 모델의 출력(클래스 점수)과 실제 정답 레이블(클래스)와의 관계를 기반으로 함
- 점수는 입력 이미지와 가중치(W) 파라미터의 함수이므로 손실 함수는 입력 이미지 x_i, 파라미터 W, 정답 레이블 y_i의 함수로도 볼 수 있음
  - $L_i = L(s, y_i)$ where $s = Wx_i$
  - $L_i = L(x_i, y_i, W)$ 혹은

손실 함수 상세

- 손실 L_i 계산을 모든 훈련세트 이미지에 대해 진행함
- 보통 이들의 평균 값을 손실 함수로 정의함
  - $L = \frac{1}{N}\sum_{i=0}^{N-1} L_i$

다중클래스 서포트 벡터 머신 손실

- 다중클래스 서포트 벡터 머신 손실(multi-class support vector machine loss)은 대표적으로 사용되는 손실 중 하나
- SVM 손실 (혹은 hinge 손실)이라고도 불림
- SVM 손실은 각 이미지에 대해 정확한 클래스가 잘못된 클래스보다 일정한 마진 Δ만큼 높은 점수를 얻도록 설정
- 즉, 정확한 클래스에 대한 점수가 높은 경우 손실이 낮게 되며, 이는 좋은 것임

다중클래스 서포트 벡터 머신 손실

- i번째 훈련세트 이미지에 대해 이미지 $x_i \in \mathbb{R}^D$와 정답 레이블 $y_i \in [0, K-1]$가 주어짐
- 점수 함수는 클래스 점수의 벡터 $f(x_i, W)$를 계산하며, 이를 s로 줄여 표현
- 예를 들어, j번째 클래스의 점수는 s의 j번째 요소
  - $s = f(x_i, W)$
  - $s_j = f(x_i, W)_j$

다중클래스 서포트 벡터 머신 손실

- 이 때, 다중클래스 SVM 손실은 다음과 같이 정의됨

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- s_{y_i}가 $s_j + \Delta$ 보다 항상 높으면 손실은 0이 됨 $(j \neq y_i)$
- 그렇지 않으면 손실이 발생
- 이 손실을 최소화하면, 자동적으로 정답 클래스에 대한 점수가 다른 클래스 점수들보다 높게 됨

다중클래스 SVM 손실 예시

- 세 개의 클래스가 점수 $s = [13, -7, 11]$을 받았고, 첫 번째 클래스가 정답 클래스 (즉, $y_i = 0$)이라 가정
- 또한, Δ는 10이라고 가정
- 다음 식을 얻게 됨

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

다중클래스 SVM 손실 예시

- 첫 번째 항을 살펴보면 -7-13+10은 음수이므로 max(0,-) 함수 사용 시 0이 됨
- 손실이 0이 된 이유는 정답 클래스 점수(13)이 잘못된 클래스 점수(-7) 보다 적어도 마진 10만큼 크기 때문 (실제로는 10보다 훨씬 큼)
- SVM 손실은 차이가 10 이상이면 더 이상 상관하지 않음
- 두 번째 항은 11-13+10을 계산하여 8을 얻으며, 이는 올바른 클래스의 점수(13)가 잘못된 클래스의 점수(11)보다 높긴 하지만 마진 10만큼 크지는 않았기 때문

다중클래스 SVM 손실 예시

- 요약: SVM 손실 함수는 올바른 클래스 y_i의 점수가 잘못된 클래스 점수보다 적어도 Δ만큼 클 것을 요구하며, 그렇지 않을 경우 손실이 누적됨

- 어떤 클래스의 점수가 빨간 영역 내부(또는 그 이상)에 있다면 손실이 누적되며, 그렇지 않을 경우 손실은 0이 됨

선형 점수 함수의 경우 SVM 손실 함수

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 점수 함수로 선형 함수를 사용하는 경우의 손실 함수는?

$$s = W x_i$$

$$s_j = w_j^T x_i$$

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

힌지 손실(hinge loss)

- o에서의 임계값 함수 $\max(0, -)$ 는 종종 힌지 손실(hinge loss)라 불림
- 힌지 손실의 제곱 형태인 $\max(0, -)^2$을 사용하는 경우도 있는데, 이는 위반된 마진에 대해 더 강한 페널티(손실)를 줌
- 제곱 형태의 힌지 손실은 일부 데이터셋에서 더 잘 작동하는 경우가 있으며, 이는 검증을 통해 결정 가능

정규화

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

- 위에서 제시한 손실 함수에는 한 가지 버그가 있음
- 모든 데이터세트 이미지를 정확하게 분류($L_i = 0$ for all i)하는 파라미터 집합 W를 가정
- 이러한 W는 유일하지 않고 많은 유사한 W가 있음
- W에 대해 손실이 0이면 λW $(\lambda > 1)$에 대해서도 손실이 0

정규화

- 이와 같은 모호성을 제거하기 위해 특정 가중치 W에 대한 선호도를 포함하고자 함
- 정규화 페널티 $R(W)$를 포함하여 손실 함수를 확장
- 일반적인 정규화 페널티는 가중치 W의 모든 성분에 제곱 페널티를 주어 큰 가중치를 방지하는 제곱 L2 norm

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

다중클래스 SVM 손실(정규화 포함)

- 정규화 함수 $R(W)$는 입력 데이터와는 무관하며 가중치에만 기반을 둠
- 정규화 페널티를 포함한 완전한 다중클래스 SVM 손실은 다음과 같음
  - 데이터 손실 + 정규화 손실

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

- 정규화 페널티 때문에 정확히 0의 손실을 달성하는 것은 불가능

다중클래스 SVM 손실(정규화 포함)

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

- 위 SVM 손실을 완전한 형태로 펼쳐서 표현

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

- 하이퍼파라미터 λ는 검증을 통해 결정됨

정규화 페널티 Q&A

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

- Q) 편향 벡터는 정규화 페널티에 포함 안 되는지?
- A) 대부분의 경우 L2 정규화는 가중치 행렬 W에만 적용됨
- Q) 딥러닝에서 여러 계층의 가중치 행렬 W가 있는 경우는?
- A) 여러 계층의 가중치 행렬의 원소 제곱들을 모두 합산함

정규화 장점

- 큰 가중치에 페널티를 줌으로써 일반화를 향상시키는 경향
- 예시)
  - 입력 이미지 벡터 $x^T = [1,1,1,1]$, 가중치 벡터 $w_a^T = [1,0,0,0]$, $w_b^T = [0.25,0.25,0.25,0.25]$
  - $w_a^T x = w_b^T x = 1$ 인데 w_a의 L2 페널티는 1이며, w_b의 L2 페널티는 0.5
  - L2 페널티에 따르면 가중치 벡터 w_b가 더 선호됨

정규화 장점

- L2 페널티가 더 작고 분산된 가중치 벡터를 선호하므로, 최종 분류기는 몇몇 입력 차원을 매우 강하게 고려하기 보다 모든 입력 차원을 적은 양으로 고려하게 됨
- 이는 분류기의 일반화 성능을 향상시키고 과적합을 줄일 수 있음

손실 함수 코드 (비벡터화)

```
def L_i(x, y, W):
  """
  unvectorized version. Compute the multiclass svm loss for a single example (x,y)
  - x is a column vector representing an image (e.g. 3073 x 1 in CIFAR-10)
    with an appended bias dimension in the 3073-rd position (i.e. bias trick)
  - y is an integer giving index of correct class (e.g. between 0 and 9 in CIFAR-10)
  - W is the weight matrix (e.g. 10 x 3073 in CIFAR-10)
  """
  delta = 1.0 # see notes about delta later in this section
  scores = W.dot(x) # scores becomes of size 10 x 1, the scores for each class
  correct_class_score = scores[y]
  D = W.shape[0] # number of classes, e.g. 10
  loss_i = 0.0
  for j in range(D): # iterate over all wrong classes
    if j == y:
      # skip for the true class to only loop over incorrect classes
      continue
    # accumulate loss for the i-th example
    loss_i += max(0, scores[j] - correct_class_score + delta)
  return loss_i
```

손실 함수 코드 (비벡터화)

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 위 식을 계산하는 코드
- $\Delta = 1$로 설정
- 입력 이미지 x: 3072 크기 벡터, W: 10 x 3072 행렬
- 편향 트릭 사용시 x: 3073 크기 벡터, W: 10 x 3073 행렬

```
def L_i(x, y, W):
  """
  unvectorized version. Compute the multiclass svm loss for a single example (x,y)
  - x is a column vector representing an image (e.g. 3073 x 1 in CIFAR-10)
    with an appended bias dimension in the 3073-rd position (i.e. bias trick)
  - y is an integer giving index of correct class (e.g. between 0 and 9 in CIFAR-10)
  - W is the weight matrix (e.g. 10 x 3073 in CIFAR-10)
  """
  delta = 1.0 # see notes about delta later in this section
```

손실 함수 코드 (비벡터화)

```
scores = W.dot(x) # scores becomes of size 10 x 1, the scores for each class
correct_class_score = scores[y]
D = W.shape[0] # number of classes, e.g. 10
```

- 먼저 점수들을 계산함
  - W와 x의 행렬-벡터 곱으로 계산 가능
  - scores: 크기 10 벡터
- 올바른 클래스의 점수 계산
  - y: 정답 레이블 y_i에 해당
  - scores[y]: s_{y_i}에 해당
- D는 클래스 개수 (10)

손실 함수 코드 (비벡터화)

```
loss_i = 0.0
for j in range(D): # iterate over all wrong classes
  if j == y:
    # skip for the true class to only loop over incorrect classes
    continue
  # accumulate loss for the i-th example
  loss_i += max(0, scores[j] - correct_class_score + delta)
return loss_i
```

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 위 코드는 바로 위 손실 식을 계산한 것

손실 함수 코드 (반벡터화)

```
def L_i_vectorized(x, y, W):
  """
  A faster half-vectorized implementation. half-vectorized
  refers to the fact that for a single example the implementation contains
  no for loops, but there is still one loop over the examples (outside this function)
  """
  delta = 1.0
  scores = W.dot(x)
  # compute the margins for all classes in one vector operation
  margins = np.maximum(0, scores - scores[y] + delta)
  # on y-th position scores[y] - scores[y] canceled and gave delta. We want
  # to ignore the y-th position and only consider margin on max wrong class
  margins[y] = 0
  loss_i = np.sum(margins)
  return loss_i
```

손실 함수 코드 (반벡터화)

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 위 손실을 계산하기 위해 먼저 다음 벡터를 계산
  - $[s_0 - s_{y_i} + \Delta,\ s_1 - s_{y_i} + \Delta, \cdots, s_9 - s_{y_i} + \Delta]$
- 이는 scores – scores[y]+delta 를 계산하면 됨 (broadcasting)

```
scores = W.dot(x)
# compute the margins for all classes in one vector operation
margins = np.maximum(0, scores - scores[y] + delta)
# on y-th position scores[y] - scores[y] canceled and gave delta. We want
# to ignore the y-th position and only consider margin on max wrong class
margins[y] = 0
loss_i = np.sum(margins)
```

손실 함수 코드 (반벡터화)

- 이제 각 성분에 임계값 함수 $\max(0, -)$ 을 적용함
  - $[\max(0, s_0 - s_{y_i} + \Delta), \max(0, s_1 - s_{y_i} + \Delta), \cdots, \max(0, s_9 - s_{y_i} + \Delta)]$
  - 이는 np.maximum(o, scores – scores[y]+delta) 에 해당 (broadcasting)

```
scores = W.dot(x)
# compute the margins for all classes in one vector operation
margins = np.maximum(0, scores - scores[y] + delta)
# on y-th position scores[y] - scores[y] canceled and gave delta. We want
# to ignore the y-th position and only consider margin on max wrong class
margins[y] = 0
loss_i = np.sum(margins)
```

손실 함수 코드 (반벡터화)

- y_i번째 항을 0으로 만듦
  - $[\max(0, s_0 - s_{y_i} + \Delta), \max(0, s_1 - s_{y_i} + \Delta), \cdots, \max(0, s_9 - s_{y_i} + \Delta)]$ 에서 y_i번째 항이 0인 벡터가 됨
  - 이는 margins[y] = 0 으로 실행 가능
- 그 후 모든 항들을 다 더하여 손실 L_i를 얻음
  - 이는 np.sum(margins)로 수행 가능

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

```
scores = W.dot(x)
# compute the margins for all classes in one vector operation
margins = np.maximum(0, scores - scores[y] + delta)
# on y-th position scores[y] - scores[y] canceled and gave delta. We want
# to ignore the y-th position and only consider margin on max wrong class
margins[y] = 0
loss_i = np.sum(margins)
```

손실 함수 코드 (완전벡터화)

```
def L(X, y, W):
  """
  fully-vectorized implementation :
  - X holds all the training examples as columns (e.g. 3073 x 50,000 in CIFAR-10)
  - y is array of integers specifying correct class (e.g. 50,000-D array)
  - W are weights (e.g. 10 x 3073)
  """
  # evaluate loss over all examples in X without using any for loops
  # left as exercise to reader in the assignment
```

- X: 3073 x 50000 행렬
- y: 크기 50000 배열
- W: 10 x 3073 행렬
- 완전히 벡터화하여 모든 loop 를 제거하여 속도를 빠르게 함

실제적인 고려사항 – 델타 설정

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 하이퍼파라미터 Δ의 값은 얼마로 설정?
- Δ 값은 1로 설정해도 충분
- Δ와 λ는 모두 데이터 손실과 정규화 손실 사이의 균형을 조절하는 동일한 역할을 함

실제적인 고려사항 – 델타 설정

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

- 점수 사이의 마진의 정확한 값(ex. $\Delta = 1$ 또는 $\Delta = 100$)은 다소 무의미함
- 가중치 W의 크기를 조절하면서 점수의 차이의 크기를 임의로 줄이거나 증가시킬 수 있기 때문
- 따라서, 정규화 강도 λ를 통해서 가중치를 얼마나 크게 허용할 것인지만 결정하면 됨

실제적인 고려사항 – 이진 SVM과의 관계

- 머신러닝에서 이진 SVM에서의 i번째 예제에 대한 손실은 다음과 같음

$$L_i = C \max(0, 1 - y_i w^T x_i) + R(W)$$

- 여기서 C는 하이퍼파라미터이며, y_i는 {-1, 1} 중 하나의 값을 가짐
- 두 클래스만 있을 경우 이 강의자료에서의 SVM은 위 이진 SVM에 해당
- C는 λ와 동일한 트레이드오프를 제어하며, C와 λ는 역수 관계로 연결됨

실제적인 고려사항 – 이진 SVM과의 관계

- 본 강의자료의 다중 클래스 SVM은 여러 클래스에 대한 SVM 형식화 방법 중 하나에 불과
- 다른 대표적인 형태는 One-Vs-All (OVA) SVM
- 이는 각 클래스에 대해 다른 모든 클래스와 대비하여 이진 SVM을 훈련시킴
- 본 강의자료에서의 형식화는 Weston과 Watkins 1999 버전을 따름
  - Support Vector Machines for Multi-Class Pattern Recognition

소프트맥스 분류기(softmax classifier)

- SVM 외에 다른 대표적인 분류기는 소프트맥스 분류기(softmax classifier)
- 이진 로지스틱 회귀 분류기(binary logistic regression classifier)의 여러 클래스에 대한 일반화
- SVM은 각 클래스에 대한 해석이 어려울 수 있는 점수를 사용하는 반면, 소프트맥스 분류기는 점수에 좀 더 직관적인 의미(확률론적)를 부여

소프트맥스 분류기(softmax classifier)

- 소프트맥스 분류기에서 점수 함수 $f(x_i; W) = Wx_i$는 동일하지만, 이 점수들을 각 클래스에 대한 비정규화된 로그 확률로 해석하고, 힌지 손실을 크로스-엔트로프 손실로 대체

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

- 비정규화된은 확률들이 적당히 스케일링 된 상태로 합이 1이 아닌 경우를 의미

확률	p_1	p_2	…	p_k
비정규화된 로그확률	$\log(Cp_1)$	$\log(Cp_2)$		$\log(Cp_k)$

소프트맥스 분류기(softmax classifier)

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

- f_j: 클래스 점수 벡터 f의 j번째 성분
- 전체 손실 L은?

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

소프트맥스 함수

- 소프트맥스 함수

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

- 이는 임의의 실수 값 점수 벡터 z를 합계가 1이 되는 벡터로 압축함

확률적 해석

확률	p_1	p_2	…	p_k
비정규화된 로그확률	$\log(Cp_1)$	$\log(Cp_2)$		$\log(Cp_k)$

- 소프트맥스 분류기에서는 점수 s_j를 비정규화된 로그 확률(j가 정답 클래스일 확률)로 해석

점수	s_1	s_2	…	s_k
확률	$\frac{e^{s_1}}{\sum_j e^{s_j}}$	$\frac{e^{s_2}}{\sum_j e^{s_j}}$	…	$\frac{e^{s_k}}{\sum_j e^{s_j}}$

확률적 해석

점수	s_1	s_2		s_k
확률	$\frac{e^{s_1}}{\sum_j e^{s_j}}$	$\frac{e^{s_2}}{\sum_j e^{s_j}}$	$\cdots$	$\frac{e^{s_k}}{\sum_j e^{s_j}}$

- 즉, $\frac{e^{s_y}}{\sum_j e^{s_j}}$를 마치 $P(y|x_i; W)$ 로 해석

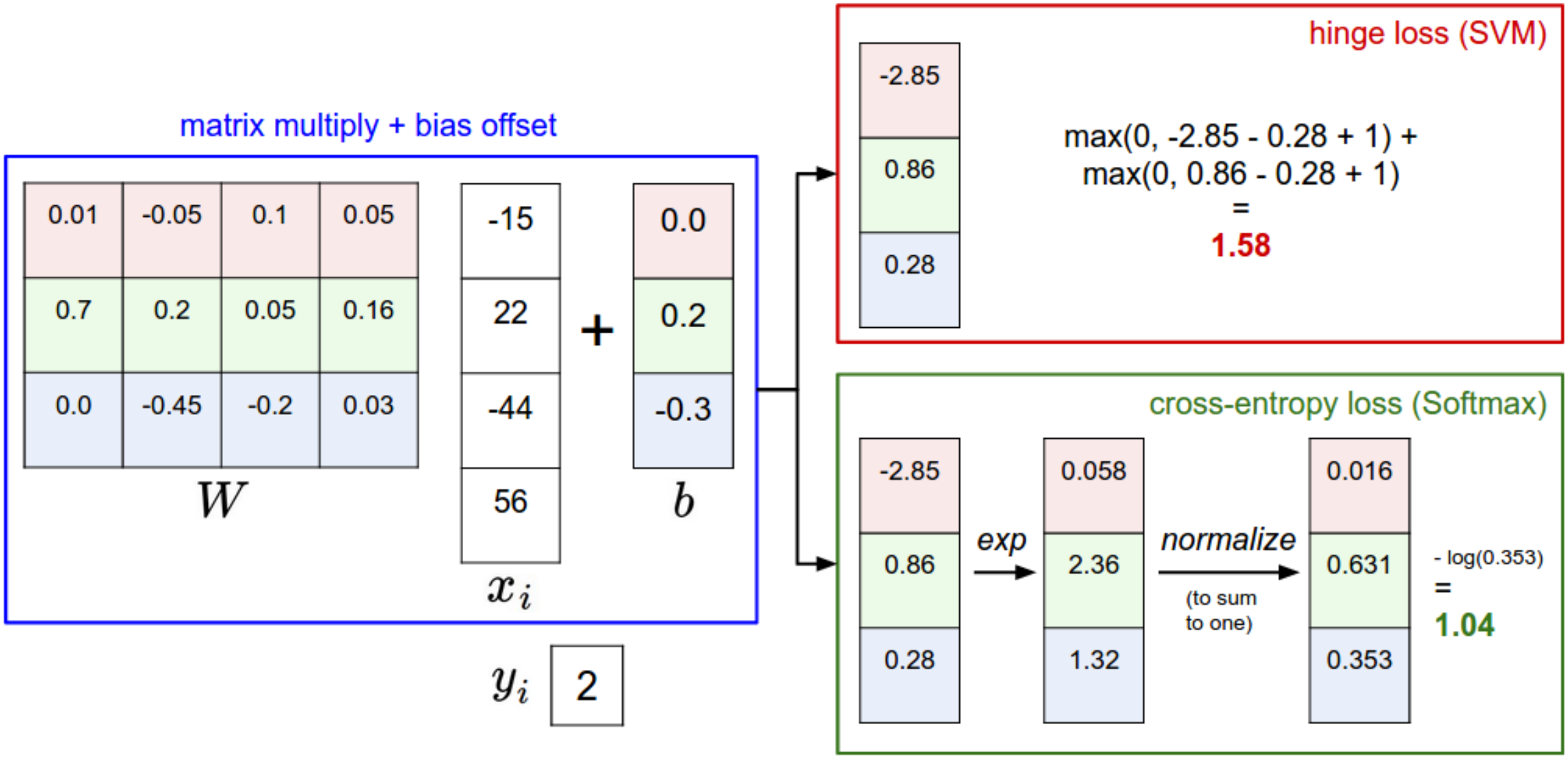
확률적 해석

- 주어진 모델에 대해 y_i가 정답 클래스일 확률은 $P(y_i|x_i;W)$
- 손실은 이 확률의 –log 값으로 정의됨

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

- 이상적으로 s_{y_i}가 무한대이고 나머지 j에 대해 s_j값들은 0이 되기를 원함
- 이 경우 손실은 0이 되며 그렇지 않으면 손실이 생김
- s_{y_i}가 다른 점수 s_j들보다 커지는 방향으로 학습될 것

SVM v.s. 소프트맥스 분류기



소프트맥스 분류기 출력

- SVM은 해석하기 어려운 점수를 계산하는 반면 소프트맥스 분류기는 각 클래스에 대한 확률을 계산함
- ex) 3가지 클래스: 고양이, 개, 배

  SVM: [12.5, 0.6, -23.0], 소프트맥스: [0.9, 0.09, 0.01]
- 확률은 각 클래스에 대한 신뢰도로 해석 가능

소프트맥스 분류기 출력

- 확률이 얼마나 뾰족하거나 얼마나 확산되어있는지는 정규화 강도 λ에 직접적으로 의존
- ex) 점수: [1, -2, 0]

λ: ↓

$$[1, -2, 0] \rightarrow [e^1, e^{-2}, e^0] = [2.71, 0.14, 1] \rightarrow [0.7, 0.04, 0.26]$$

λ: ↑

$$[0.5, -1, 0] \rightarrow [e^{0.5}, e^{-1}, e^0] = [1.65, 0.37, 1] \rightarrow [0.55, 0.12, 0.33]$$

SVM v.s. 소프트맥스 분류기 (성능)

- SVM과 소프트맥스 간 성능 차이는 일반적으로 매우 작음
- SVM은 더 지역적인 목표를 가짐
- ex) SVM($\Delta = 1$)에서 점수가 [10,-2,3]이고 첫 번째 클래스가 올바른 클래스인 경우 생각
- 이미 다른 클래스 점수에 비해 마진보다 높은 점수를 가지므로 손실이 0으로 계산됨
- 즉, SVM은 개별 점수의 세부 사항에는 관심이 없음

SVM v.s. 소프트맥스 분류기 (성능)

- 점수가 [10, -100, -100]이든 [10, 9, 9]이든 SVM은 손실이 0이라고 판단하고 무관심
- 소프트맥스 분류기의 경우 [10, 9, 9]에 훨씬 더 높은 손실을 누적
- 즉, 소프트맥스 분류기는 생산되는 점수에 결코 만족하지 않으며, 손실은 항상 개선될 수 있음