

Lecture 2 | Image Classification

자주 사용할 python module : numpy

<https://cs231n.github.io/python-numpy-tutorial/>

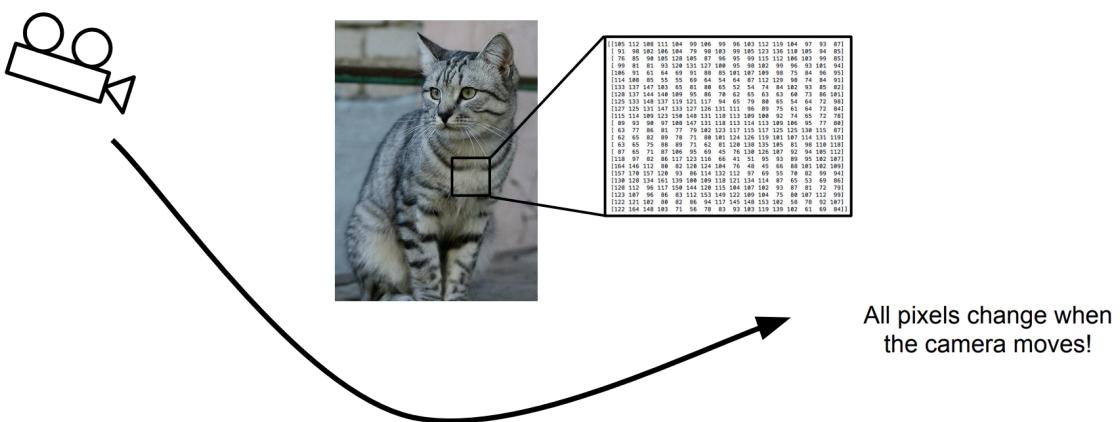
앞으로 할 것 : Image Classification (이미지 분류)

컴퓨터가 그림을 인식할 때, 각 픽셀을 단위로 하여 3가지 정수 값(RGB)으로 인식
앞으로, 이 값을 이용하여 이미지를 분류할 것임.

문제점 (이미지 분류의 어려움)

▼ 시점의 변화 (Viewpoint variation)

아래 그림처럼, 같은 장면을 다른 시점에서 찍은 사진이라면 다른 데이터로 인식할 것임.



▼ 조명 (Illumination)

밝은 경우, 어두운 경우 등 조명의 상태에 따라 사진의 RGB 데이터에 많은 변화가 생김.



▼ 변형 (Deformation)

대상의 자세, 형태 등에 따라 데이터 상의 많은 차이가 발생함.



▼ 폐색 (Occlusion)

아래 그림처럼 얼굴만 찍혔거나, 풀숲에 가려지거나, 꼬리만 나와있는 경우 이미지 인식이 상당히 어려움.



▼ 배경의 혼잡함 (Background Clutter)

대상이 어떤 배경에 있는가도 데이터에 많은 영향을 끼침



▼ 같은 종류 끼리의 다양함 (Intraclass variation)

"고양이"라고 해서 모두 동일하지 않음. 색, 형태, 크기 등이 굉장히 다양함을 고려하고 분류할 수 있어야 함.



Data-Driven Approach

우리가 이미지 분류를 할 방법은 다음과 같다.

1. 분류할 대상에 대한 이미지들의 데이터 셋을 모은다.
2. 기계학습(Machine learning)을 사용하여 미리 수집한 데이터 셋을 학습시킨다.

3. 학습시킨 분류기(Classifier)으로 새로운 이미지를 분류한다.

First Classifier - Nearest Neighbor

분류하고자 하는 이미지를 입력받은 후, 학습한 데이터 셋 중 이와 가장 유사한 이미지를 찾아 동일한 라벨로 분류하는 방법

```
def train(images, labels):
    # Machine learning!
    #모든 데이터와 라벨을 저장
    return model

def predict(model, test_images):
    #Use model to predict labels
    #test_image(입력)와 가장 유사한 데이터를 찾아 이와 동일한 라벨로 분류
    return test_labels
```

Example Dataset : CIFAR10

<http://www.cs.toronto.edu/~kriz/cifar.html>

10개의 라벨이 있고, 각 라벨당 5000개의 training images, 1000개의 test images가 존재한다.

즉, 50000개의 사진을 학습시켜서 10000개의 사진을 분류하려는 것이 목적이다.

사진 비교할 때 사용할 것 : Distance Metric

$$L1 \text{ Distance} : d(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

각 픽셀 별 RGB, 즉 각각의 데이터들의 모든 차이를 계산하고 그 차이들의 전체 합으로 거리를 계산한다.

test image				training image				pixel-wise absolute value differences			
56	32	10	18	10	20	24	17	46	12	14	1
90	23	128	133	8	10	89	100	82	13	39	33
24	26	178	200	12	16	178	170	12	10	0	30
2	0	255	220	4	32	233	112	2	32	22	108

- = add → 456

L1 Distance를 구현하여 CIFAR10을 분류한 전체 코드

```

import numpy as np

class NearestNeighbor(object):
    def __init__(self):
        pass

    def train(self, X, y):
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        num_test = X.shape[0]
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        for i in xrange(num_test):
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances)
            Ypred[i] = self.ytr[min_index]
        return Ypred

Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/')
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3)
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3)
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
Yte_predict = nn.predict(Xte_rows)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )

```

코드 분석

```

Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/')
# Xtr, Ytr : training images' datas & labels

```

```

# Xte, Yte : test images' datas & labels
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3)
# CIFAR10은 값들을 나열해둔 것이므로 가로 * 세로 * 3(RGB) 크기로 reshape함.
# 간단하게 말하면, 모두 묶여있는 데이터 나열을 그림별로 묶는다는 의미
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3)
# 분류할 데이터도 위와 동일하게 reshape

```

코드 상에 주석으로 달아 놓았지만, 간단하게 설명을 덧붙이자면 읽어들인 데이터셋은 숫자들의 나열에 불과하다. 우리는 이 숫자들의 나열을 이용하여 그림들을 분류해야하기 때문에, 그림 별로 데이터를 묶어주려고 한다. 그림 하나당 데이터 $32 \times 32 \times 3$ (가로 * 세로 * 3(RGB))개 이므로 reshape를 이용하여 그림 별로 묶어준다.

Training Data(Xtr)과 Test Data(Xte) 모두 동일 작업을 시행해준다.

```

nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# Training Data 를 이용한 학습
Yte_predict = nn.predict(Xte_rows)
# Test Data의 라벨 구분
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )

```

'NearestNeighbor'라는 클래스(추후 설명할 것임)를 선언해 주고, 클래스 내부 함수인 train과 predict를 이용하여 정확도를 체크하는 코드이다.

```

class NearestNeighbor(object):
    def __init__(self):
        pass

    def train(self, X, y):
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        num_test = X.shape[0]
        # X.shape => [사진의 개수 , 가로*세로*3]
        # 아까 위에서 reshape 해주었기 때문에 위와 같은 결과가 나옴.
        # X.shape[0] = 사진의 개수
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        for i in xrange(num_test):
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            # np.abs(self.Xtr - X[i,:])
            # 위처럼 계산하면, Broadcasting에 의해 모든 데이터와 비교가 가능
            min_index = np.argmin(distances)
            # 저장되어있는 거리를 중 최소를 찾아주는 numpy 내부의 argmin()함수를 사용

```

```

Ypred[i] = self.ytr[min_index]
return Ypred

```

'NearestNeighbor' 라는 클래스에 대한 설명이다.

우선, train 함수에서는 입력한 data&label을 저장한다.

predict 함수에서는 num_test에 test할 데이터 개수를 저장하고, Ypred라는 배열을 0으로 초기화하여 만든다.

그리고 반복문을 이용하여 각각의 test data를 이전에 저장해놓은 training data와 비교하여 가장 작은 값을 YPred에 인덱스 별로 저장한다.

최종적으로 YPred에는 test data 별 예측한 라벨이 저장되어 있을 것이다.

Result

위의 코드를 실행시켜보면, 정확도가 38.6%밖에 되지 않는다.

Time complexity

train : O(1)

predict : O(N)

다른 거리 계산 방법

위 코드에서는 L1-Distance로 사진 별 거리(데이터 차이)를 계산하였다. 이 방법 말고도 L2-Distance를 고려해볼 수 있다.

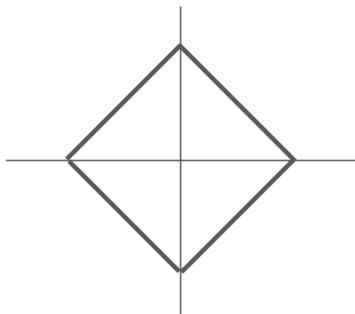
$$L2\ Distance : d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

기하학적으로 보았을 때, L1 Distance는 각각의 축 별로 거리의 합으로 Distance를 계산하는 반면, L2 Distance는 기하학적인 거리로 Distance를 계산한다고 볼 수 있다.

아래 그림은 원점에서 같은 거리에 있는 점들의 집합인 '원'을 그린 그래프인데, 좌측 그림은 L1 Distance를 기준으로, 우측 그림은 L2 Distance를 기준으로 나타낸 것이다.

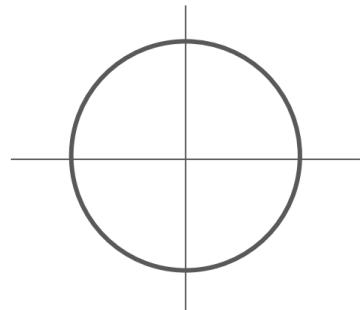
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



L2 (Euclidean) distance

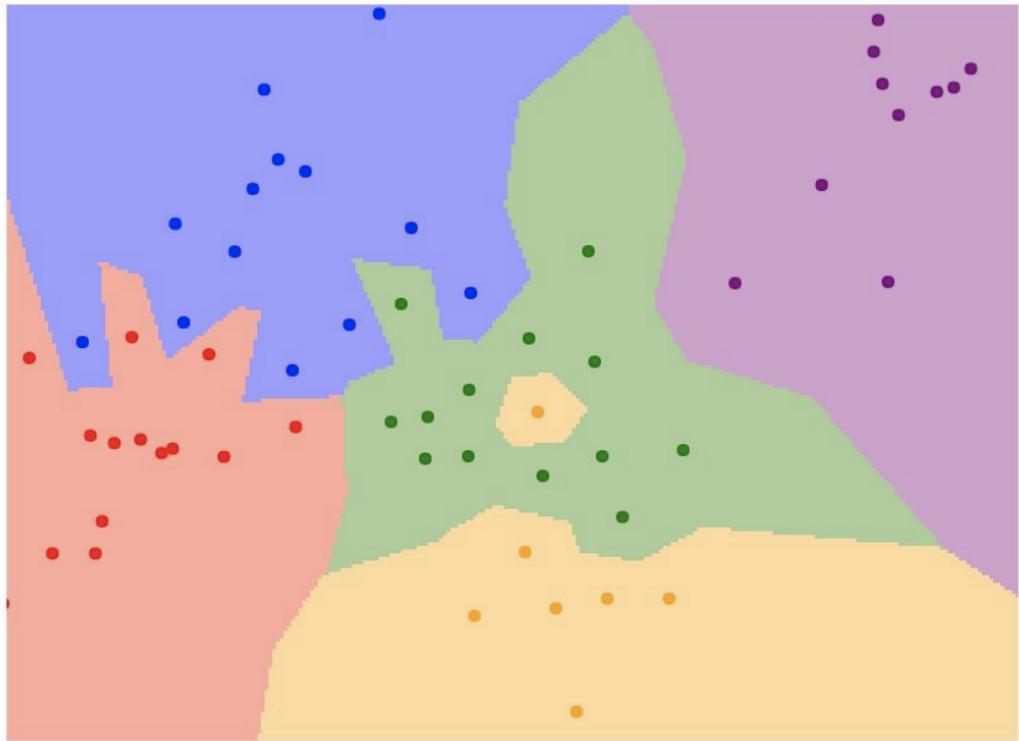
$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



K-Nearest Neighbors (KNN)

Nearest Neighbor을 이용하여 분류해보니 38.6%라는 좋지 않은 정확도로 이미지를 분류하였다. 그래서 고안한 방법이 K-Nearest Neighbors이다. 이는 가장 유사한 이미지, 즉 거리가 가까운 1개의 이미지를 찾는 것이 아니라, 유사한 k 개의 이미지를 찾고 그중 가장 많은 비중의 라벨로 분류하는 방법이다.

아래의 사진을 보자. 점들이 training data이며 동일한 색이 곧 동일한 라벨을 의미한다. 영역별 색은 해당 위치의 데이터가 들어 왔을 때, 우리가 어떤 라벨로 인식할 것인지에 대한 색이다. 아래의 그림은 위에서 설명한 Nearest Neighbors, 즉 $k=1$ 인 상황에 대한 그림이다.

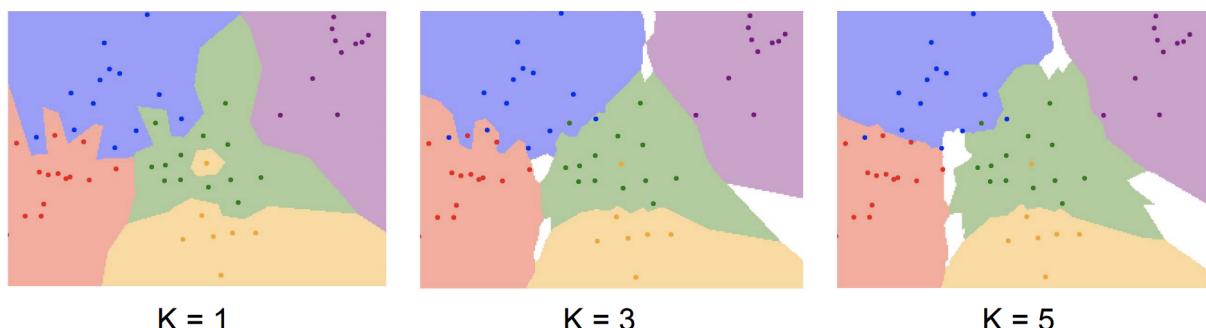


그런데 조금 이상한 부분들이 있다. 정 가운데에 노란색 training data가 있다보니 그 주변의 일부 영역이 노란색으로 표시된다. 즉, 해당 영역의 데이터는 노란색에 해당하는 라벨로 분류한다는 것이다. 그렇기에 영역들의 경계가 굉장히 이상하게 생겼다.

전체적으로 그림을 보았을 때, 가운데에 있는 노란색 데이터는 특이 데이터라고 볼 수 있다. 이런 데이터들이 존재하기 때문에 Nearest Neighbors로 분류하였을 때의 정확도가 상당히 낮게 나오는 것이다.

정 가운데의 노란색 근처 영역의 데이터를 분류하려고 할 때, 노란색 라벨보다는 초록색 라벨로 분류하는 것이 정확도 상 높을 것임을 예측할 수 있다. 그렇기 때문에 가까운 1개의 데이터가 아닌, 가까운 k 개의 데이터를 이용하여 분류를 진행하려고 한다.

아래의 $k=3$, $k=5$ 일 때의 상황을 보면 k 가 증가할수록, 가장 가까운 데이터 여러개를 비교하다 보니 영역들의 경계가 더욱 확실해짐을 알 수 있다



그렇다고 k 가 클수록 좋은 것은 아니다. 상황에 따라 적합한 k 가 있을 것이다. k 뿐만 아니라, 상황에 따라 적합한 Distance 계산 방법(L1 Distance, L2 Distance 등등)도 있을 것이다.

이와 같이 데이터 셋을 학습시키기 전에 우리가 정하는 변수들이 여럿 있는데 이들을 Hyperparameter라고 한다.

그렇다면 적합한 Hyperparameter를 어떻게 설정해야 할까?

Setting Hyperparameter k

여기서 주의해야 할 점은, 우리가 k 를 정할 때는 training data만 알고, test data는 모르고 있다는 것이다. 즉, training data만을 가지고 어떻게 k 를 설정할지 알아볼 것이다.

아이디어 1 : 가장 높은 정확도를 보이는 k 로 설정

이 방법으로 하면 무조건 $k=1$ 이 된다. 왜냐하면, training data 전체를 교육시킨 후 training data로 테스트해야하기 때문에 $k=1$ 일 때 정확도가 100%로 나온다.

아이디어 2 : 전체 데이터를 training data와 test data로 구분

이렇게 하면 k 를 정하기 전에 우리는 test data를 모른다는 전제를 가지게 된다. 즉, training data로 학습시킨 결과가 test data에서는 어떻게 작동할지 절대 예측할 수 없다.

test data는 아래에서도 계속 등장할텐데, test data를 이용하여 테스트를 진행하는 것이 k 를 검증하는 최종 단계이며, 검증 후에는 그 전단계로 되돌아가는 것이 불가능함을 알고 있어야 한다.



아이디어 3 : 전체 데이터를 training data, validation data, test data로 구분

위의 아이디어는 k 를 지정하여 training data를 입력하였을 때 그것이 잘 작동할지 예측할 수 있는 방법이 없는 것이 문제이다. 그래서 test하기 전, validation data를 이용하여 잘 작동하는지 확인해본다면 조금 더 나을 것이다.

train	validation	test
-------	------------	------

아이디어 4 : 전체 데이터를 여러 조각과 test data로 구분

전체 데이터를 여러 조각과 test data로 구분한다. 그 후, 각각의 조각별로 조각 하나를 validation data, 나머지 조각들을 training data라고 생각하고 아이디어 3과 동일하게 시행한다. 이렇게 하면 여러번 검증이 가능하기 때문에 보다 높은 정확도를 가지는 k를 찾아낼 수 있다.

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

K-Nearest Neighbors(KNN)의 장단점

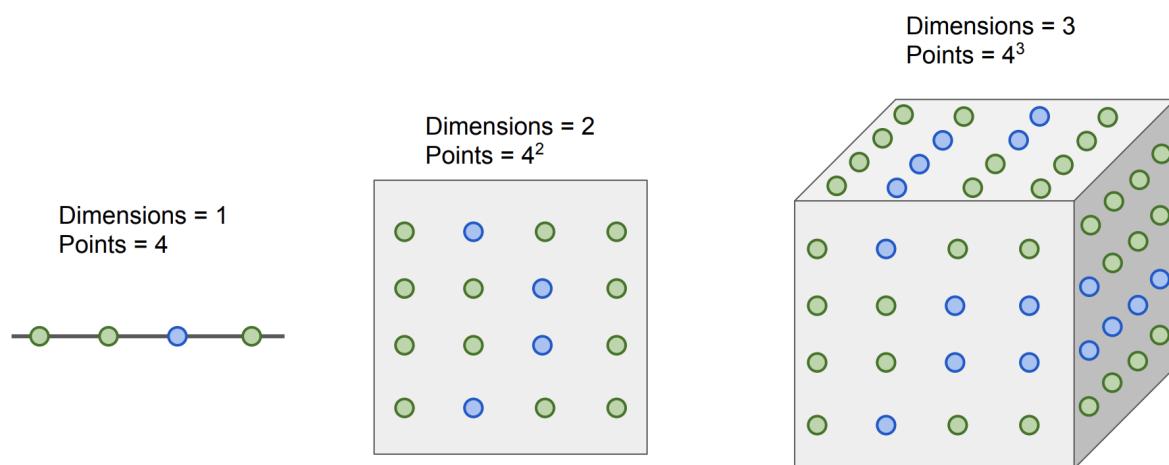
우선 학습 데이터를 그대로 저장하고 있으면 되기 때문에 학습 시간이 짧다는 장점이 있다. 반대로 실제로 분류할 때는 상당한 시간이 소요된다는 단점이 있다.

여기서 우리가 알아야 할 것은 실제 상황에서 학습 시간이 짧은 것은 아무런 이득이 없다는 것이다. 학습은 분류기를 사용하기 이전에 하는 것이므로 실제 기능에는 영향을 끼치지 않는다. 이에 반해 실제 실행 시간은 분류 시간에 영향을 받는다. 즉, 우리가 바라는 분류기는 학습시간이 길더라도 짧은 시간안에 분류해내는 분류기이다.

또, KNN은 실제로 활용하기에 무리가 있다. "거리"라는 개념은 굉장히 직관적인 개념이다. 우리가 시각적으로 바라보는 "유사한 그림"이라고 하여도 굉장히 다른 결과를 내뱉는 것이 바로 거리이다. 아래의 그림을 보면 원본 사진이 있고 이를 약간 변형시켜 눈코입을 가린 사진, 살짝 이동시킨 사진, 배경 색을 변화시킨 사진이 있다. 우리가 시각적으로 아래의 4개의 그림을 바라보면 상당히 비슷하다고 느낀다. 하지만 거리를 기준으로 보면 상당히 다르게 계산될 것이다.



뿐만 아니라, 아래의 그림처럼 거리라는 개념은 낮은 차원에서는 어느정도 가능하겠지만, 차원이 높아질수록 그 거리라는 개념은 직관적인 개념이 아니게 된다.



KNN 정리

이미지 분류(Image Classification)에서는 training set을 이용하여 학습을 하고, 최종적으로 test set의 분류를 예측한다.

KNN은 유사한 데이터들을 이용하여 예측 분류한다.

거리 재는 방법과 k가 hyperparameters이다.

validation set을 만들어 hyperparameter를 선정하자. 이 과정에서 "test set"은 가장 마지막에만 사용할 수 있음을 생각하자.

선형 분류(Linear Classification)

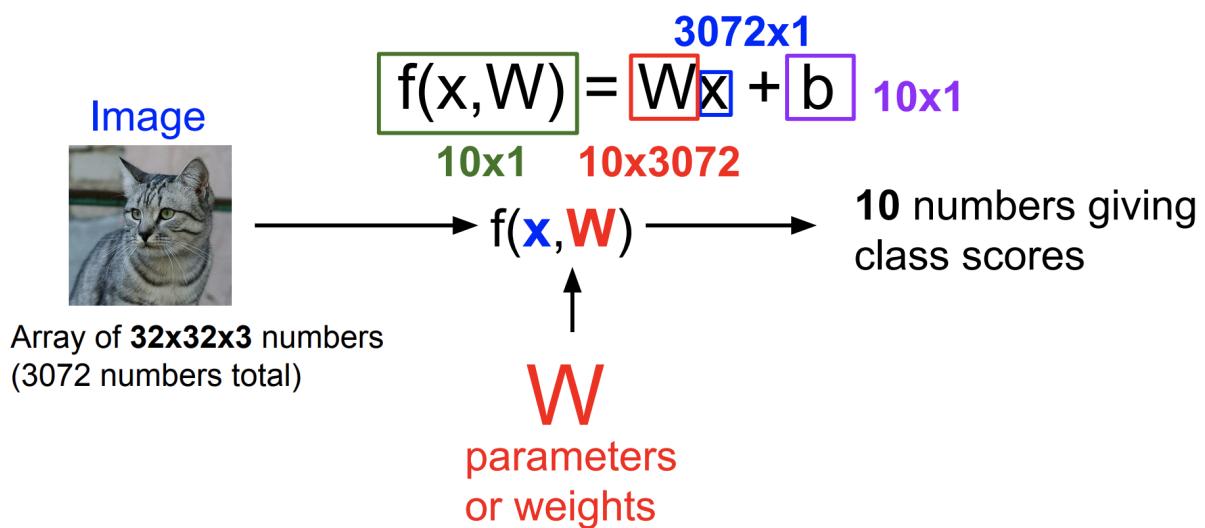
위에서 배운 KNN은 단점이 많아 실제로 사용하기에는 무리가 있었다.

이번에는 거리라는 개념 대신 이미지와 점수를 대응시키는 함수를 만들어 분류하고자 한다. 위와 동일하게 CIFAR10 데이터로 예시를 들 것이다. 이미지 하나에, 각각의 라벨에 대한 점수, 즉 10개의 점수를 얻어내는 함수를 만들 것이다. 나온 점수 중, 가장 높은 점수를 가진 라벨이 곧 주어진 그림에 대해 우리가 예측할 라벨이다.

이번 파트에서 우리가 사용할 함수는 가장 단순한 선형 함수이다.

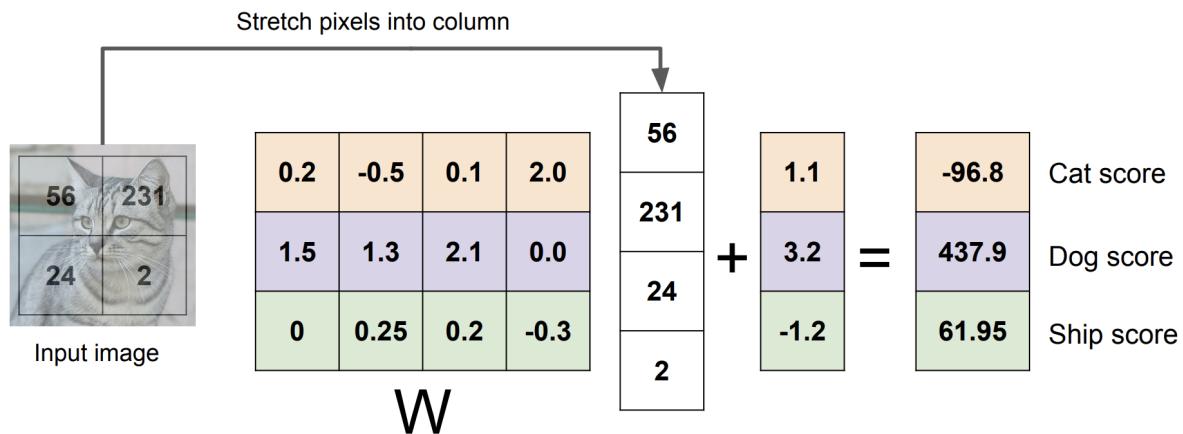
$$f(x, W, b) = Wx + b$$

아래 그림에서 보면 $f(x, W)$ 라는 함수를 볼 수 있다. x 는 우리가 예측할 이미지 데이터, W 는 우리가 사용할 매개변수, 혹은 가중치이다. 또, b 는 bias 벡터라고 하는데, 입력값에 관련없이 결과에 영향을 준다.

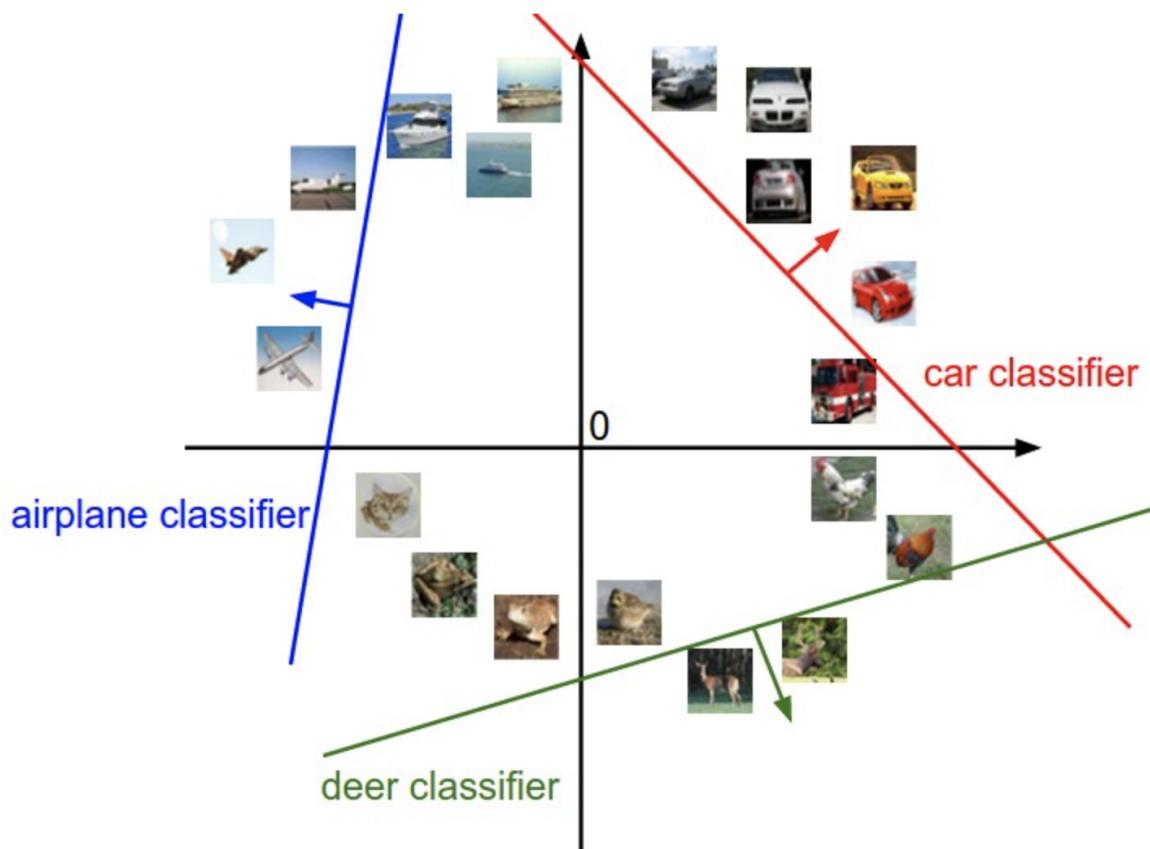


실제 계산하는 예시를 살펴보자.

아래의 그림처럼 예측할 이미지에서 데이터를 받아내고, W 에 곱한 이후 b 를 더하면 각각의 라벨에 대한 점수를 얻는다.



이를 시각화 해보면, 각 라벨에 대한 W 와 b 가 존재하기 때문에 고차원상의 직선 $y = Wx + b$ 로 표현할 수 있다.



그러면 예를 들어보자.

3개의 그림을 입력으로 하여 선형 분류기에 넣어 각각 라벨 별로 점수들을 얻게 된다. 첫 번째 이미지는 고양이인데, 점수 상으로 보면 강아지의 점수가 가장 높게 나타나고 분류기는 이 사진을 강아지로 분류할 것이다. 자동차 이미지의 경우 자동차의 점수가 가장 높은 것으로 보아 올바르게 분류될 것이다. 개구리 이미지는 개구리 라벨에 대한 점수는 상당히 낮고 트럭 라벨의 점수가 높은 것으로 보아 트럭으로 분류될 것이다.



airplane	-3.45	-0.51	3.42
automobile	-8.87	6.04	4.64
bird	0.09	5.31	2.65
cat	2.9	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	-4.34
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

이처럼 W 와 b 를 올바르게 설정하지 않으면 정확성이 상당히 떨어지게 된다. 그렇다면 어떻게 하면 정확도를 높이는 W 와 b 를 찾을 수 있을까? 이는 다음 단원에서 알아보도록 하자.