

# Image Classification

CS5670: Intro to Computer Vision

April 13

Abe Davis

# References

- Stanford CS231N
  - <http://cs231n.stanford.edu/>

# Image Classifiers in a Nutshell

- Input: an image
- Output: the class label for that image
  
- Label is generally one or more of the discrete labels used in training
  - e.g. {cat, dog, cow, toaster, apple, tomato, truck, ... }

```
def classifier(image):  
    //Do some stuff  
    return class_label;
```

$$f\left(\img alt="A ginger tabby cat sitting down." data-bbox="622 412 702 537">\right) = \text{"Cat"}$$

$$f\left(\img alt="A small dog, possibly a Corgi, sitting on a white surface." data-bbox="624 593 704 719">\right) = \text{"Dog"}$$

$$f\left(\img alt="A silver and black toaster." data-bbox="624 783 702 904">\right) = \text{"Toaster"}$$

# The Semantic Gap



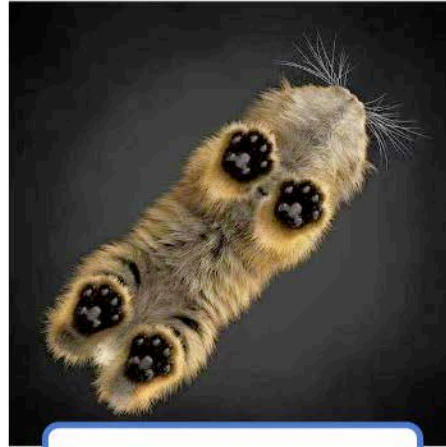
What we see

```
)1110 111010010111  
11010111 1101010101  
1 01001 11110101110  
101010111101110110  
11001111001 1 01 10  
1010111 11111011010  
101 11101 100100101  
10011111 1000111001  
1010010010011100011  
10000 1000111011100  
0110000001111 1 10  
) 1011001 010011 10  
11 10001 1111 1011  
)111 101010 10111 1  
1000101010010101101  
1001 1111000010 110  
)0111100001111 1101
```

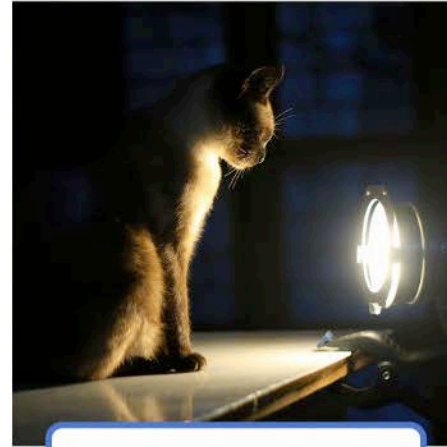
What the computer sees

# Variation Makes Recognition Hard

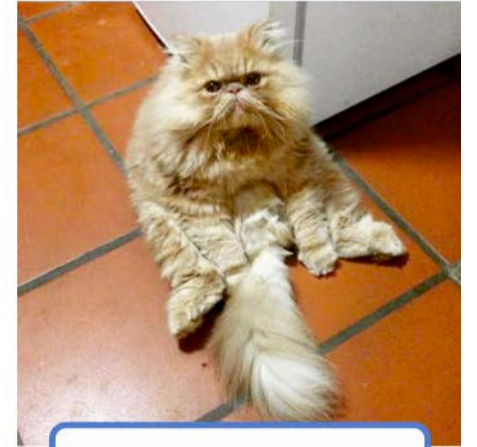
- The same class of object can appear *very* differently in different images



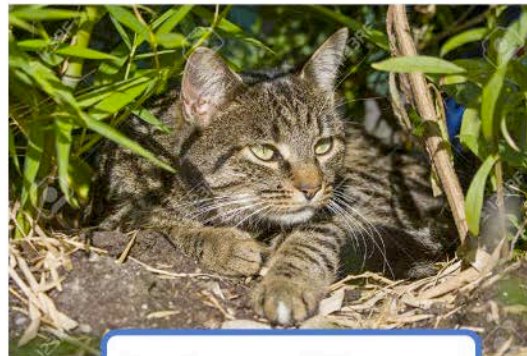
Viewpoint Variation



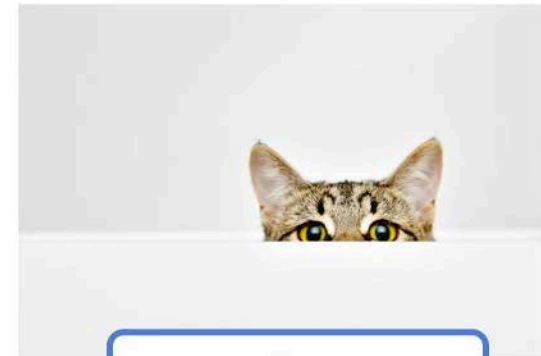
Lighting Variation



Deformation



Background Clutter



Occlusion

# The Problem is Under-constrained

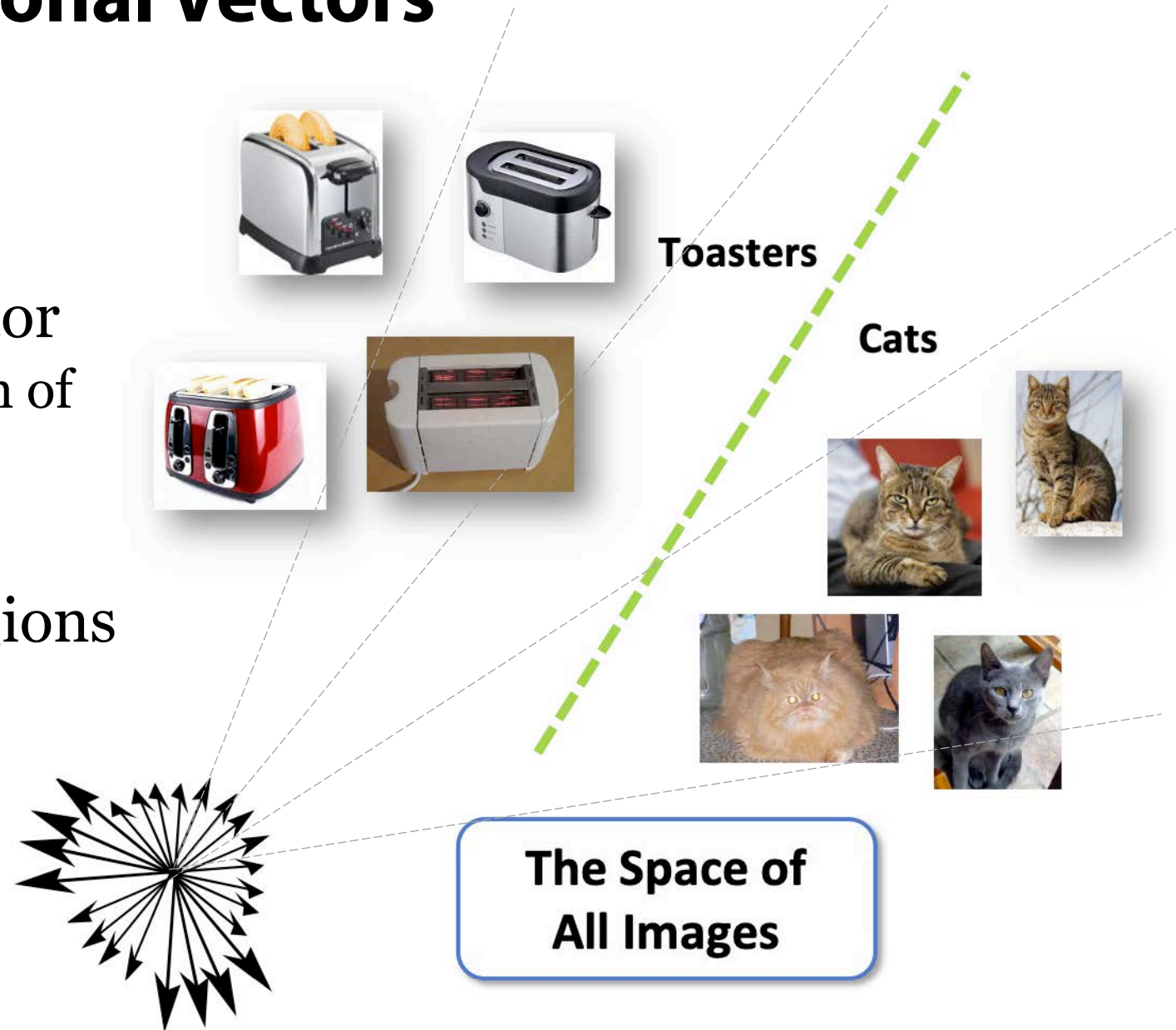
- Distinct realities can produce the same image...
- We generally can't compute the "right" answer, but we can compute the most likely one...
- We need some kind of prior to condition on. We can learn this prior from data:

$$f(x) = \underset{\ell_x}{\operatorname{argmax}} P(\ell_x | \text{data})$$



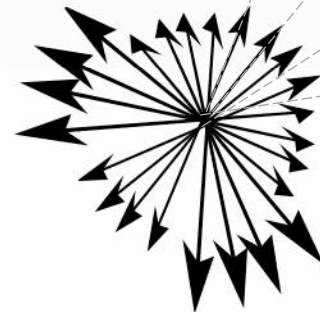
# Images As High-Dimensional Vectors

- An image is just a bunch of numbers
- Let's stack them up into a vector
  - Our training data is just a bunch of high-dimensional points now
- Divide space into different regions for different classes



# Images As High-Dimensional Vectors

- An image is just a bunch of numbers
- Let's stack them up into a vector
  - Our training data is just a bunch of high-dimensional points now
- Divide space into different regions for different classes



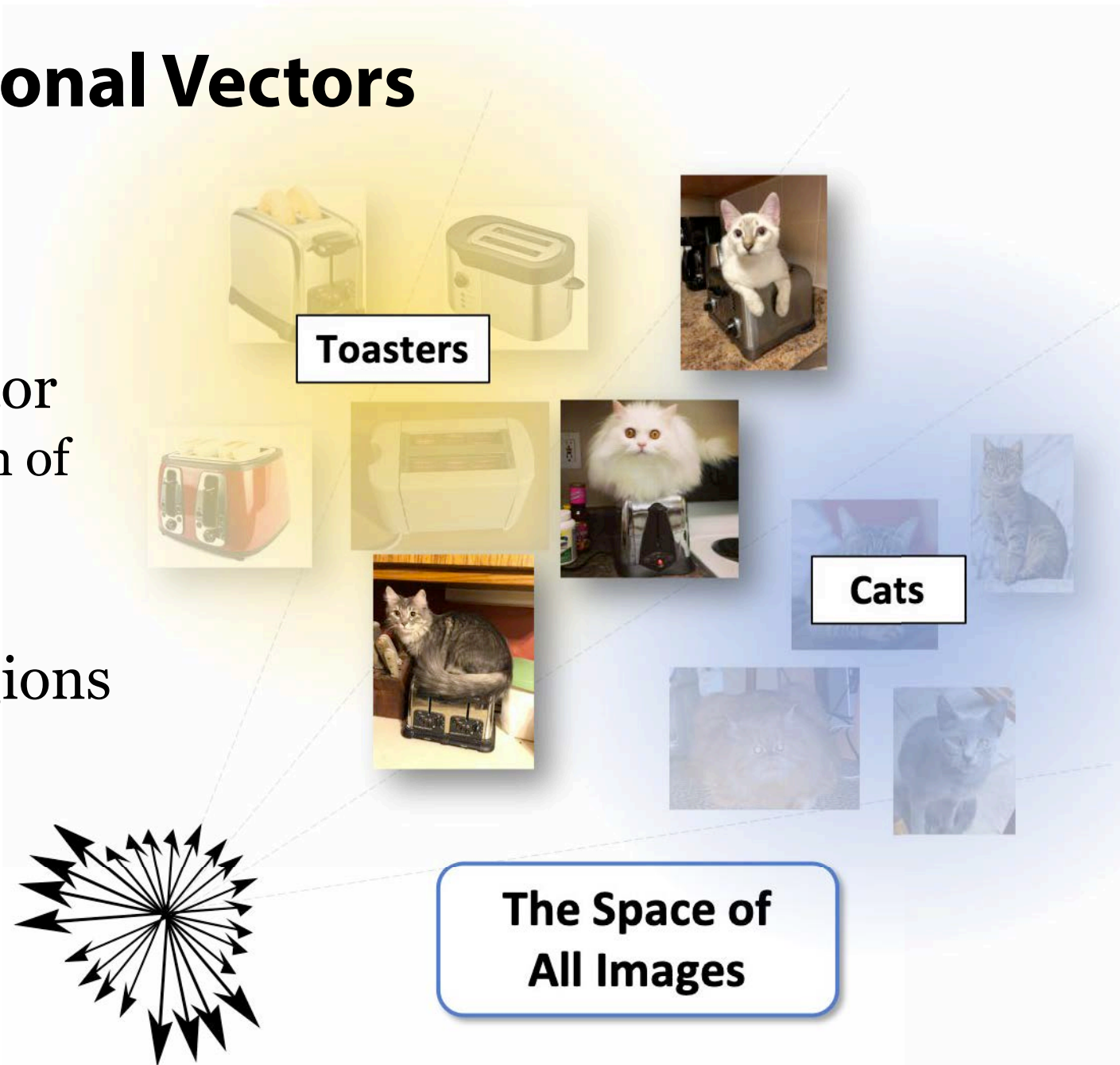
**The Space of  
All Images**





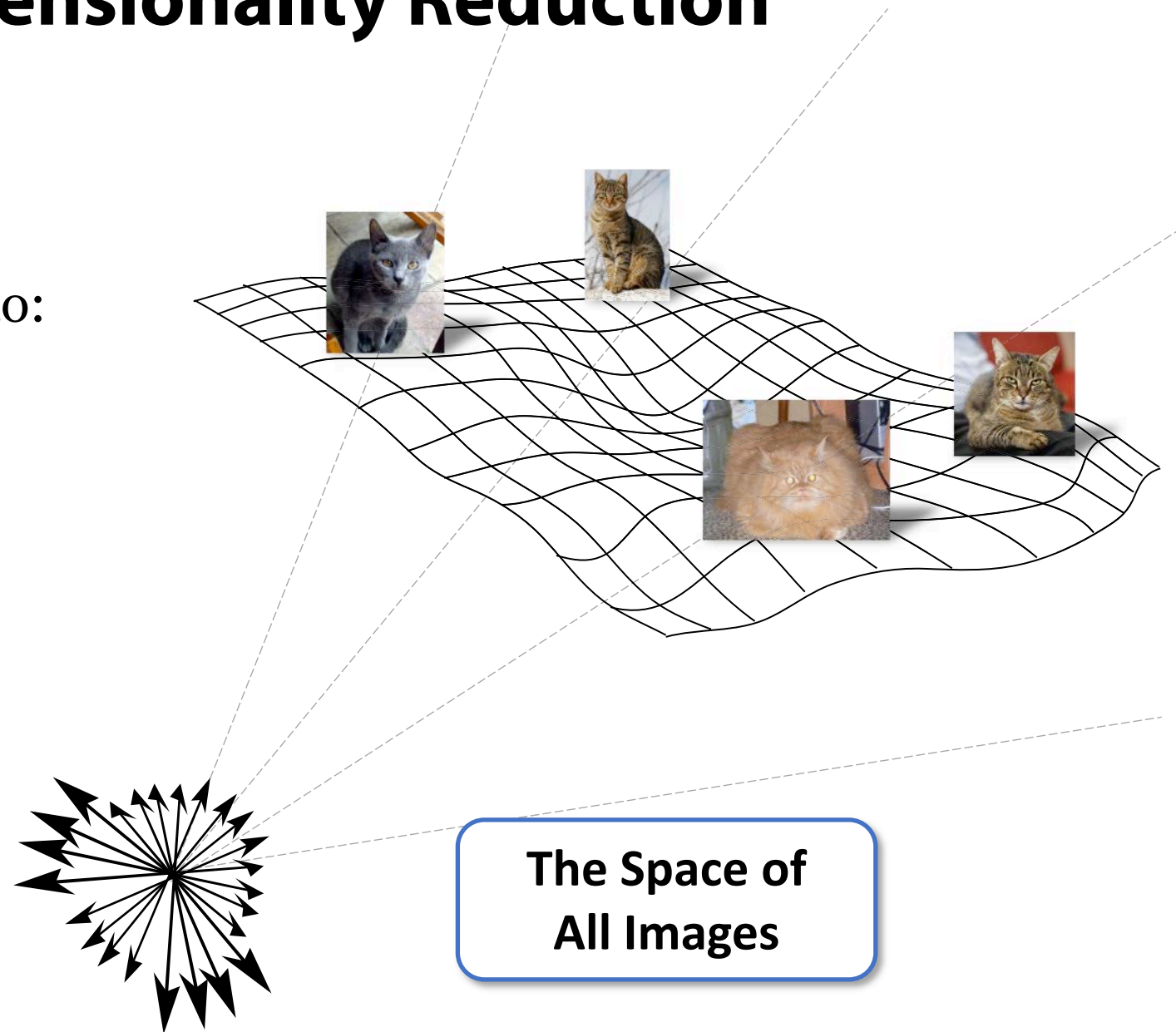
# Images As High-Dimensional Vectors

- An image is just a bunch of numbers
  - Let's stack them up into a vector
    - Our training data is just a bunch of high-dimensional points now
  - Divide space into different regions for different classes
- or**
- Define a distribution over space for each class



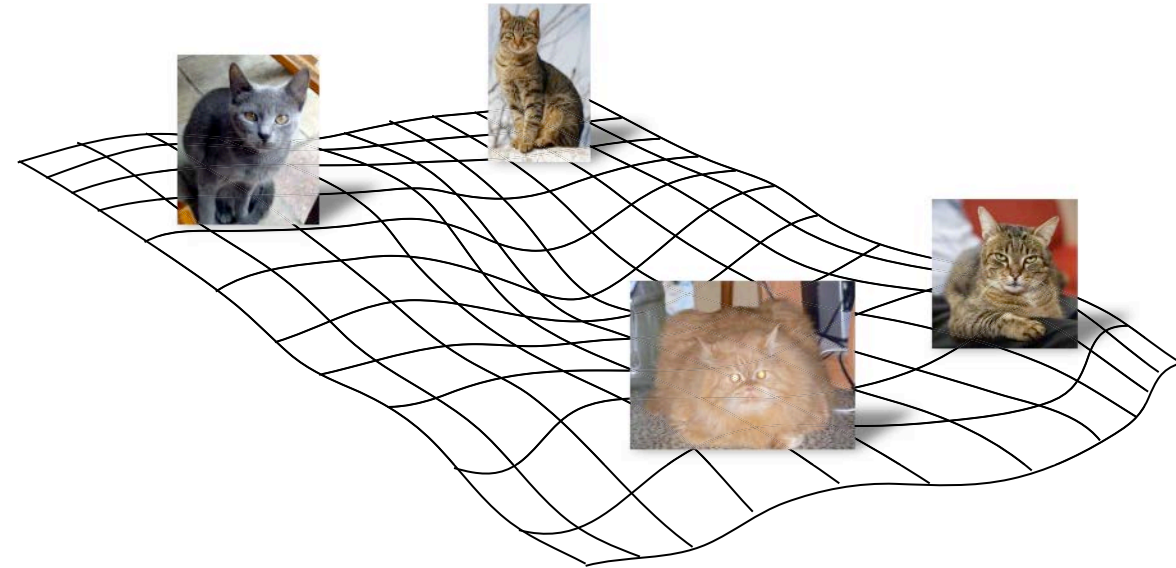
# Image Features and Dimensionality Reduction

- How high-dimensional is an image?
  - Let's consider an iPhone X photo:
    - 4032 x 3024 pixels
    - Every pixel has 3 colors
    - 36,578,304 pixels (36.5 Mega pixels)
- In practice, images sit on a lower-dimensional manifold
- Think of image features and dimensionality reduction as ways to represent images by their location on such manifolds



# Image Features and Dimensionality Reduction

- How high-dimensional is an image?
  - Let's consider an iPhone X photo:
    - 4032 x 3024 pixels
    - Every pixel has 3 colors
    - 36,578,304 pixels (36.5 Mega pixels)
- In practice, images sit on a lower-dimensional manifold
- Think of image features and dimensionality reduction as ways to represent images by their location on such manifolds



**Side Note:**  
This also lets us deal with images of different sizes, crops, etc.

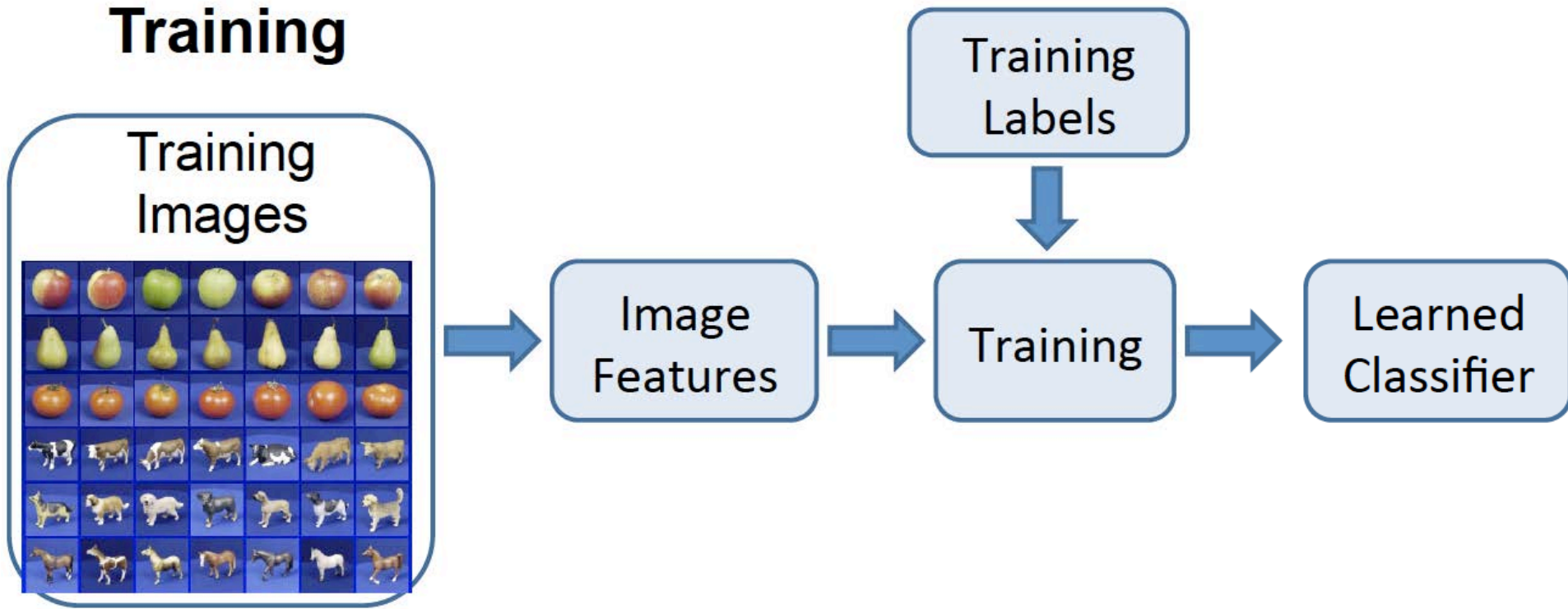
# Training & Testing a Classifier

- Collect a database of images with labels
- Use ML to train an image classifier
- Evaluate the classifier on test images

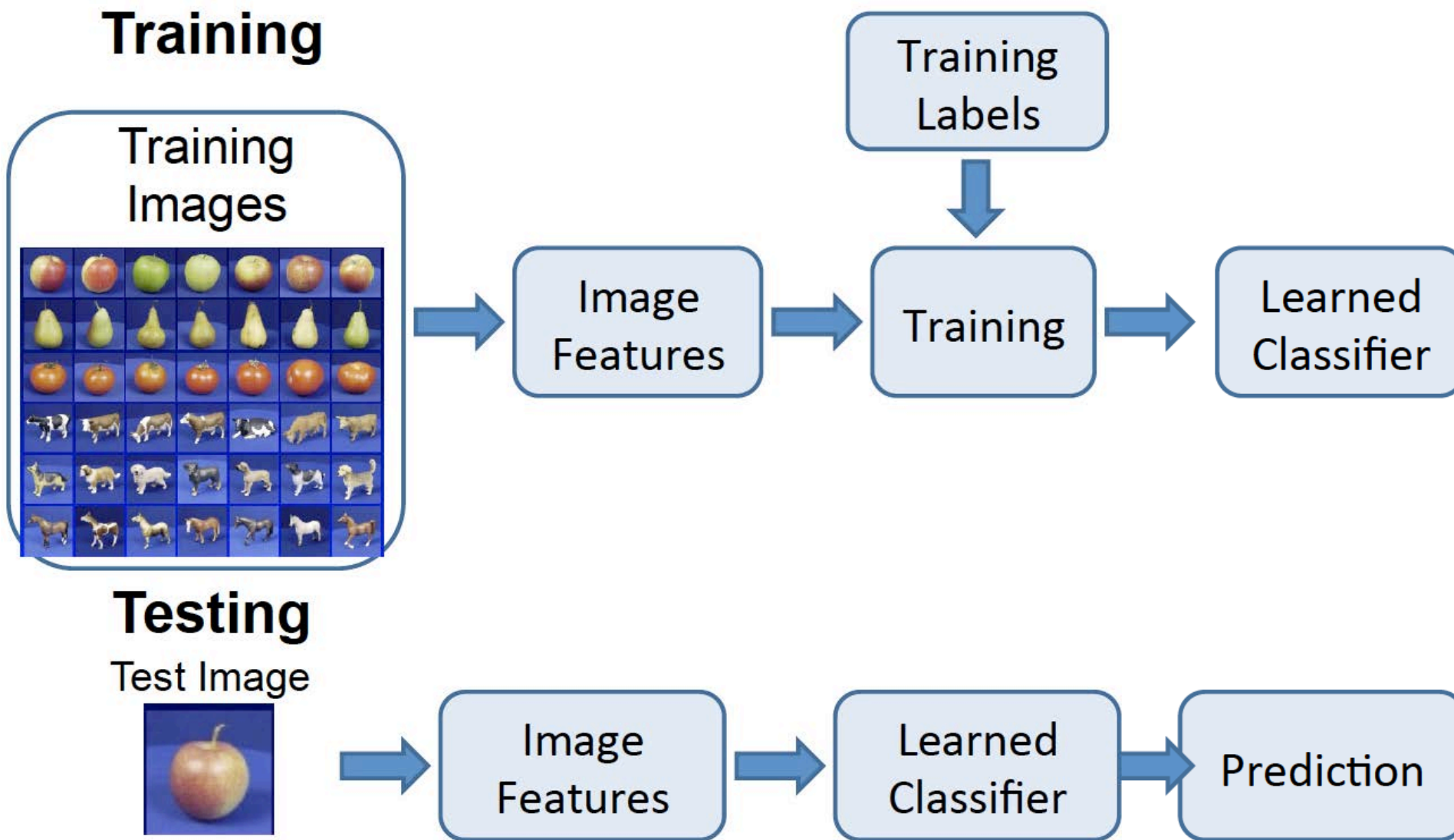
Example training set



# Training & Testing a Classifier



# Training & Testing a Classifier

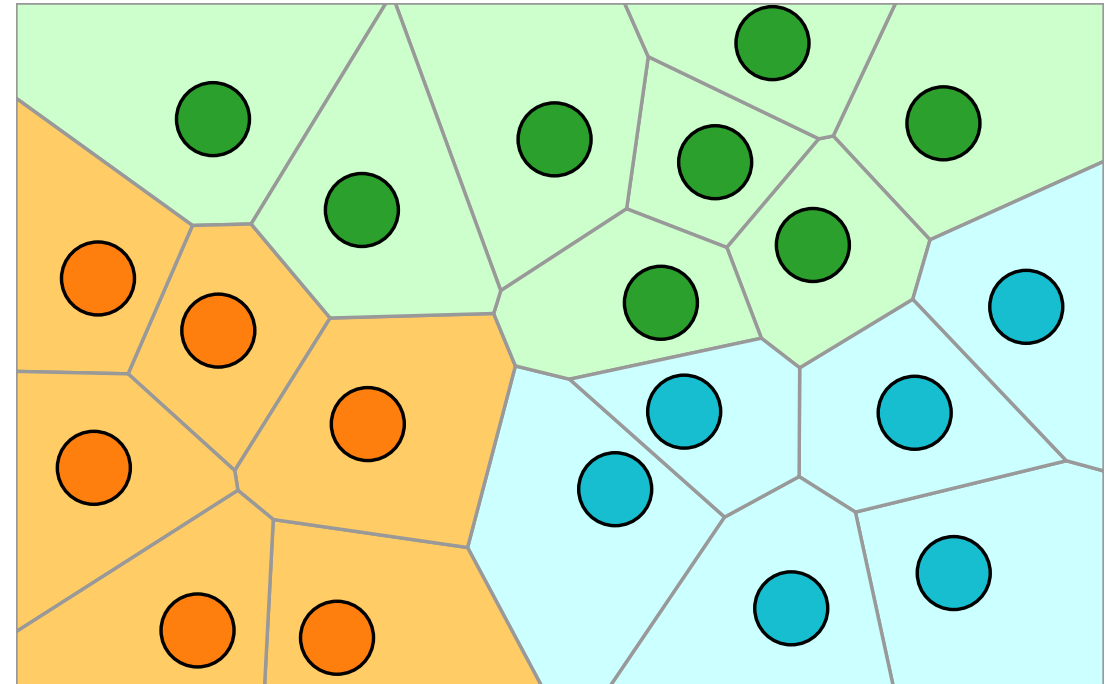


# Classifiers

- Nearest Neighbor
- kNN (“k-Nearest Neighbors”)
- Linear Classifier
- Neural Network
- Deep Neural Network
- ...

# First: Nearest Neighbor (NN) Classifier

- Train
  - Remember all training images and their labels
- Predict
  - Find the closest (most similar) training image
  - Predict its label as the true label





# CIFAR-10 and NN results

Example dataset: **CIFAR-10**

**10** labels

**50,000** training images

**10,000** test images.



# CIFAR-10 and NN results

Example dataset: **CIFAR-10**

**10** labels

**50,000** training images

**10,000** test images.



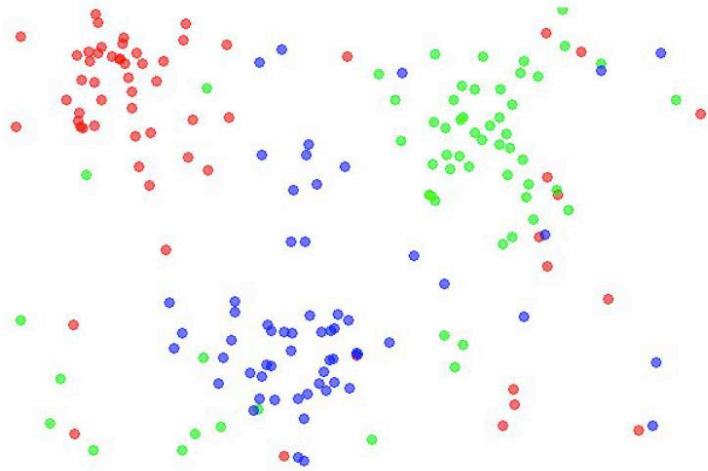
For every test image (first column),  
examples of nearest neighbors in rows



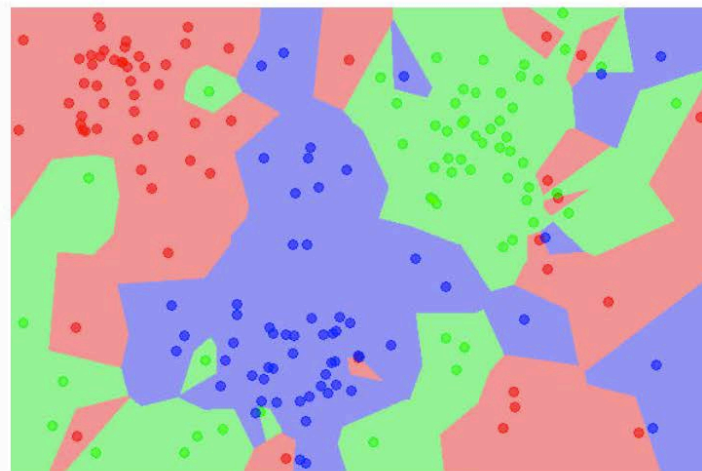
# k-nearest neighbor

- Find the  $k$  closest points from training data
- Take **majority vote** from  $K$  closest points

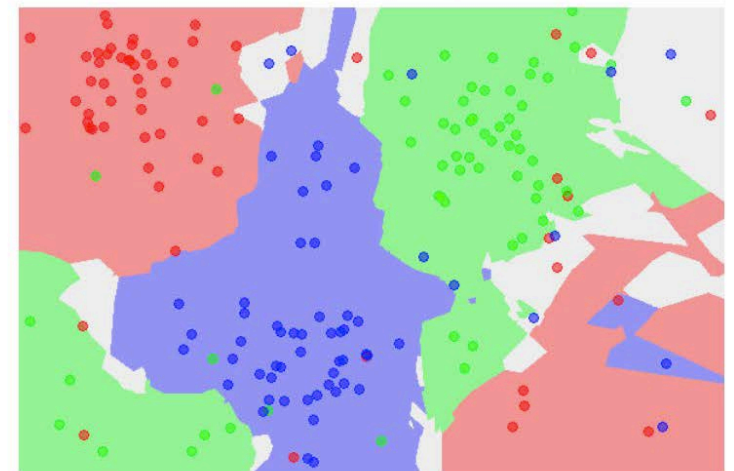
the data



NN classifier



5-NN classifier



What does this look like?





# What does this look like?



# How to Define Distance Between Images

**L1 distance:**

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

Where  $I_1$  denotes image 1,  
and  $p$  denotes each pixel

test image

|    |    |     |     |
|----|----|-----|-----|
| 56 | 32 | 10  | 18  |
| 90 | 23 | 128 | 133 |
| 24 | 26 | 178 | 200 |
| 2  | 0  | 255 | 220 |

training image

|    |    |     |     |
|----|----|-----|-----|
| 10 | 20 | 24  | 17  |
| 8  | 10 | 89  | 100 |
| 12 | 16 | 178 | 170 |
| 4  | 32 | 233 | 112 |

pixel-wise absolute value differences

|    |    |    |     |
|----|----|----|-----|
| 46 | 12 | 14 | 1   |
| 82 | 13 | 39 | 33  |
| 12 | 10 | 0  | 30  |
| 2  | 32 | 22 | 108 |

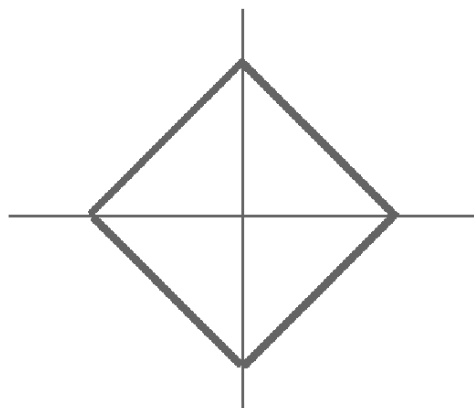
→ 456

# Choice of distance metric

- Hyperparameter

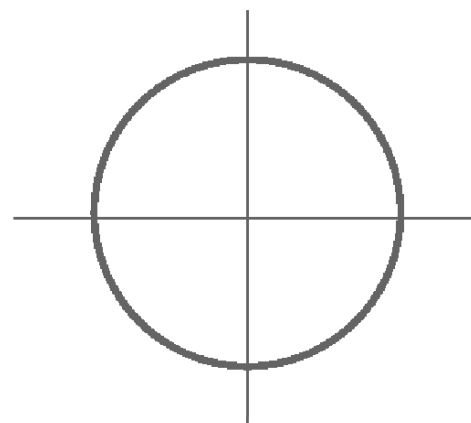
L1 (Manhattan) distance

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



L2 (Euclidean) distance

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



- Two most commonly used special cases of p-norm

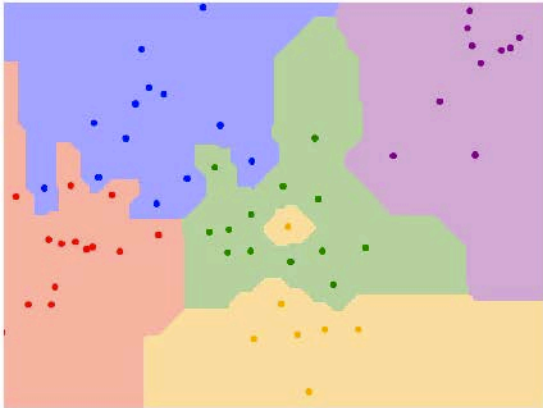
$$\|x\|_p = (|x_1|^p + \dots + |x_n|^p)^{\frac{1}{p}} \quad p \geq 1, x \in \mathbb{R}^n$$



# K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

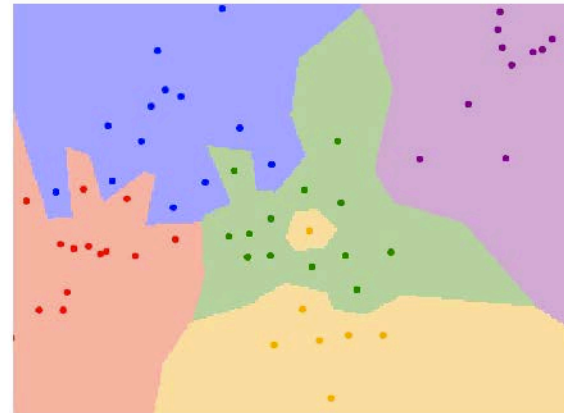
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



K = 1

L2 (Euclidean) distance

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



K = 1


Demo: <http://vision.stanford.edu/teaching/cs231n-demos/knn/>

# Hyperparameters

- What is the **best distance** to use?
- What is the **best value of k** to use?
  
- These are **hyperparameters**: choices about the algorithm that we set rather than learn
  
- How do we set them?
  - One option: try them all and see what works best

# Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data



Your Dataset

# Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

**BAD:**  $K = 1$  always works perfectly on training data

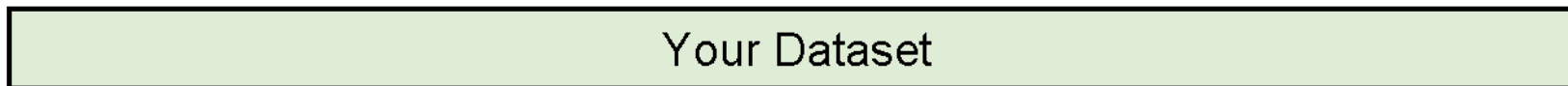


Your Dataset

# Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

**BAD:**  $K = 1$  always works perfectly on training data



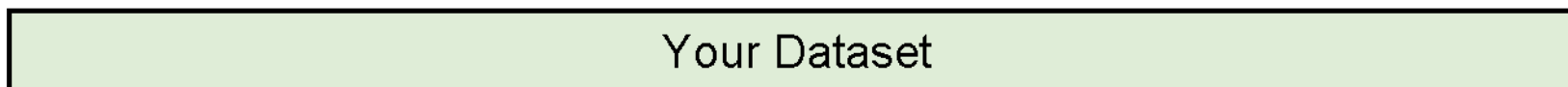
**Idea #2:** Split data into **train** and **test**, choose hyperparameters that work best on test data



# Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

**BAD:**  $K = 1$  always works perfectly on training data



**Idea #2:** Split data into **train** and **test**, choose hyperparameters that work best on test data

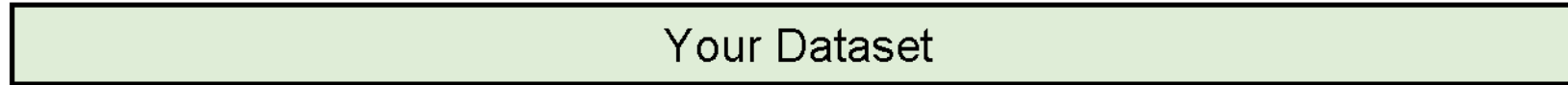
**BAD:** No idea how algorithm will perform on new data



# Setting Hyperparameters

**Idea #1:** Choose hyperparameters that work best on the data

**BAD:**  $K = 1$  always works perfectly on training data



**Idea #2:** Split data into **train** and **test**, choose hyperparameters that work best on test data

**BAD:** No idea how algorithm will perform on new data



**Idea #3:** Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test

**Better!**



# Setting Hyperparameters

Your Dataset

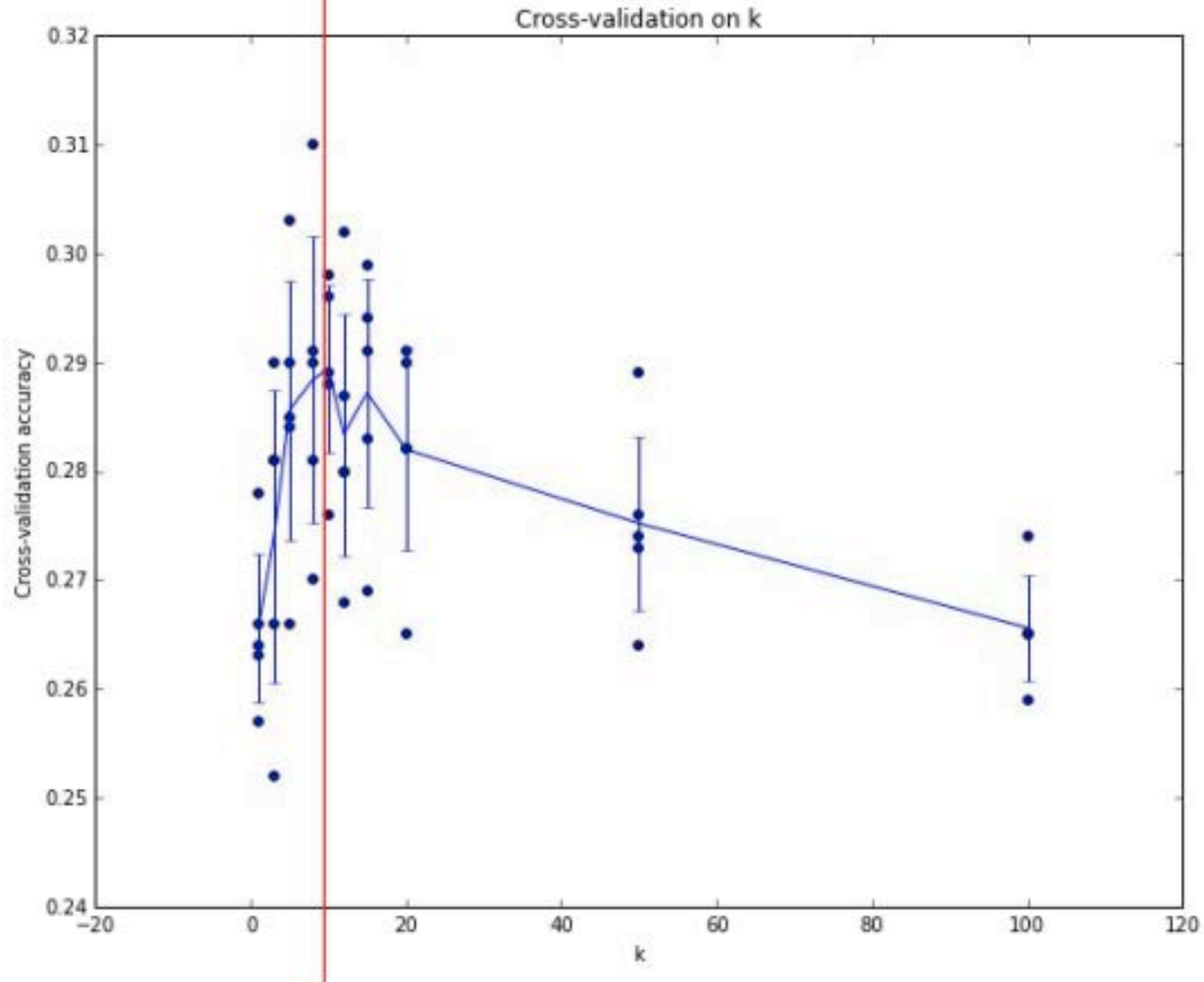
**Idea #4: Cross-Validation:** Split data into **folds**, try each fold as validation and average the results

|        |        |        |        |        |      |
|--------|--------|--------|--------|--------|------|
| 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 |

Useful for small datasets, but not used too frequently in deep learning



# Hyperparameter Tuning



Example of 5-fold cross-validation for the value of  $k$ .

Each point: single outcome.

The line goes through the mean, bars indicated standard deviation

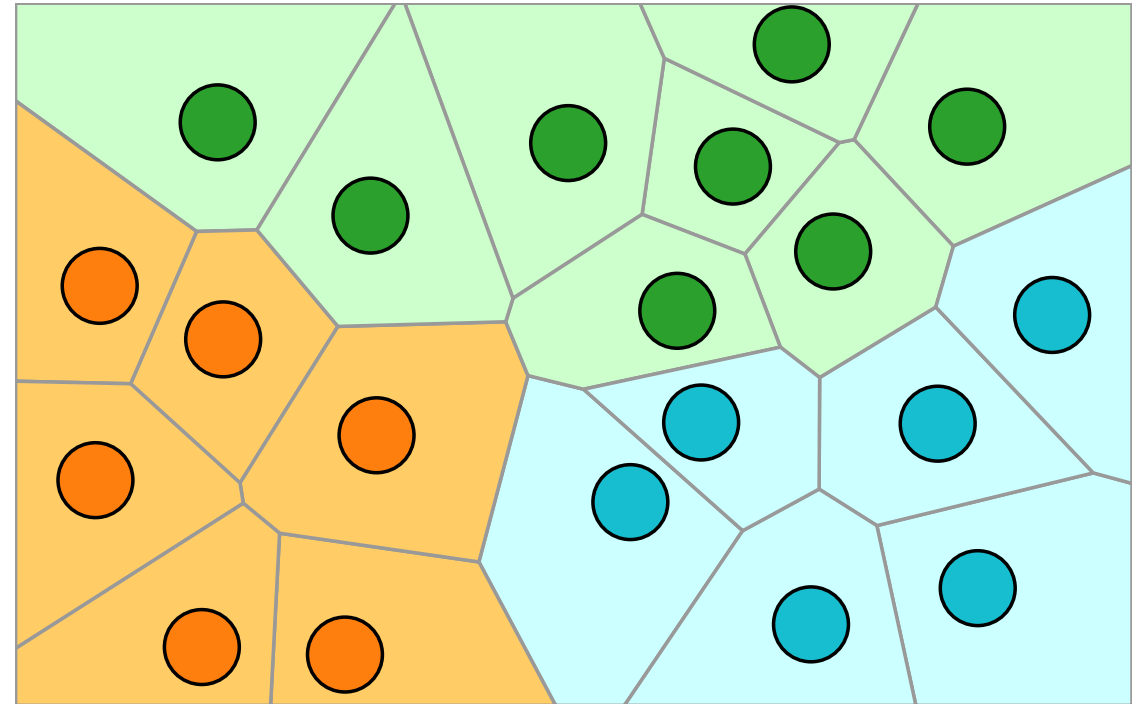
(Seems that  $k \approx 7$  works best for this data)

# Recap: How to pick hyperparameters?

- Methodology
  - Train and test
  - Train, validate, test
- Train for original model
- Validate to find hyperparameters
- Test to understand generalizability

# kNN -- Complexity and Storage

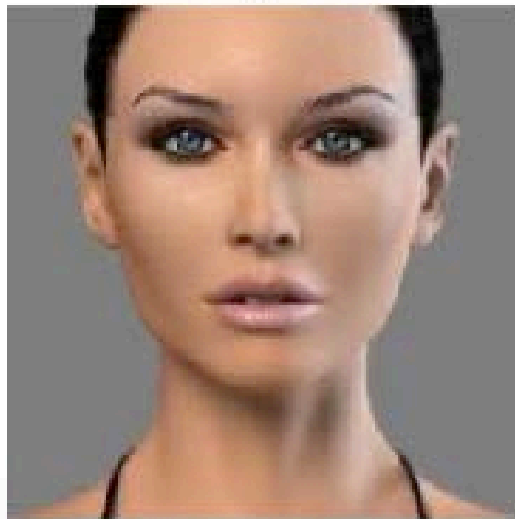
- N training images, M test images
- Training:  $O(1)$
- Testing:  $O(MN)$
- We often need the opposite:
  - Slow training is ok
  - Fast testing is necessary



# Problems with KNN: Distance Metrics

- terrible performance at test time
- distance metrics on level of whole images can be very unintuitive

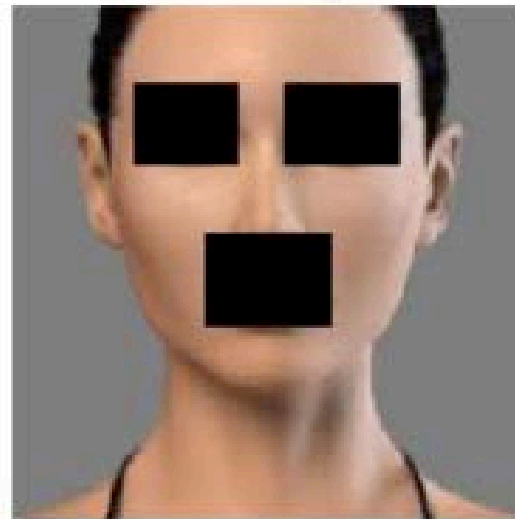
original



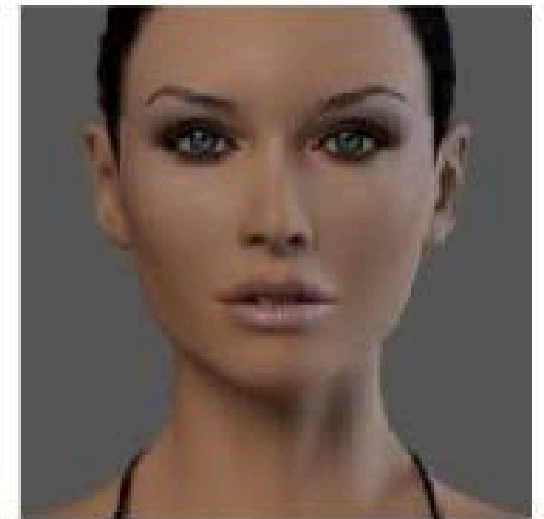
shifted



messed up



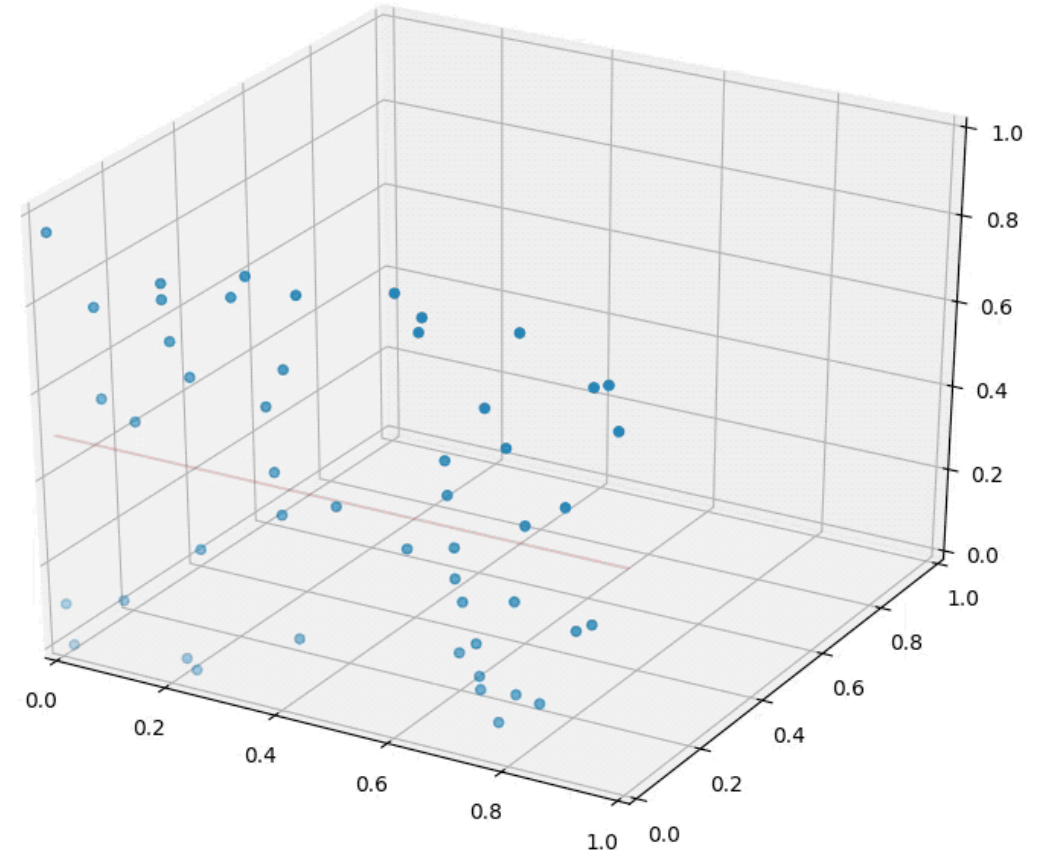
darkened



(all 3 images have same L2 distance to the one on the left)

# Problems with KNN: The Curse of Dimensionality

- As the number of dimensions increases, the same amount of data becomes more sparse.
- Amount of data we need ends up being exponential in the number of dimensions



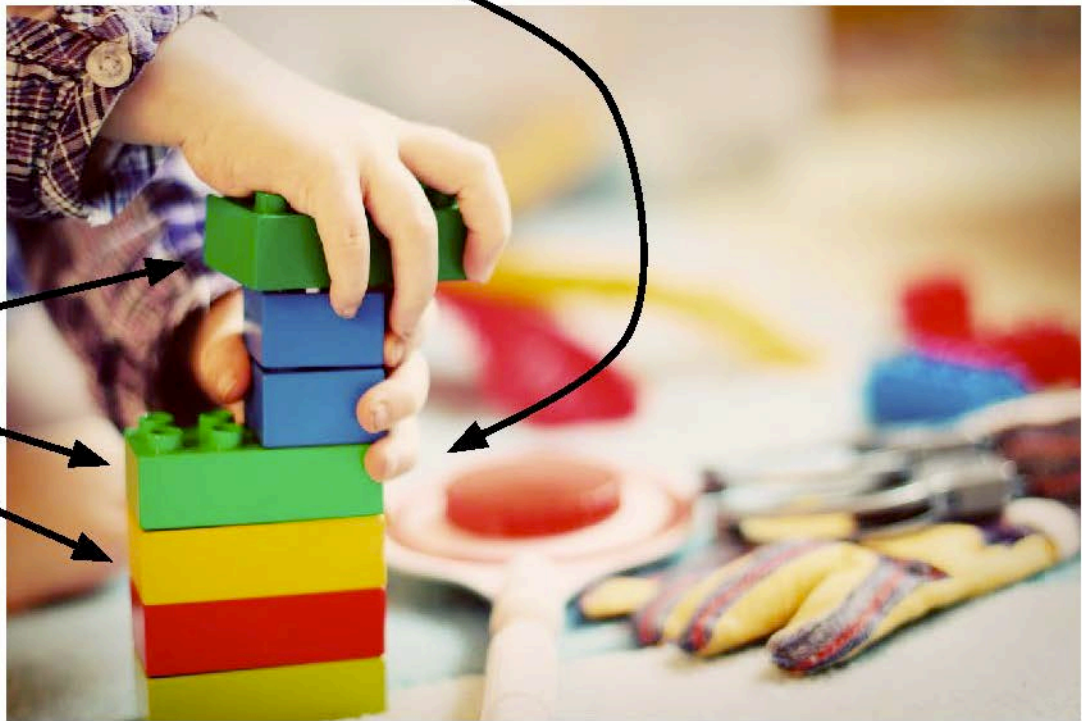
# k-Nearest Neighbors: Summary

- In **image classification** we start with a **training set** of images and labels, and must predict labels on the **test set**
- The **K-Nearest Neighbors** classifier predicts labels based on nearest training examples
- Distance metric and K are **hyperparameters**
- Choose hyperparameters using the **validation set**; only run on the test set once at the very end!

# Linear classifiers

Neural Network

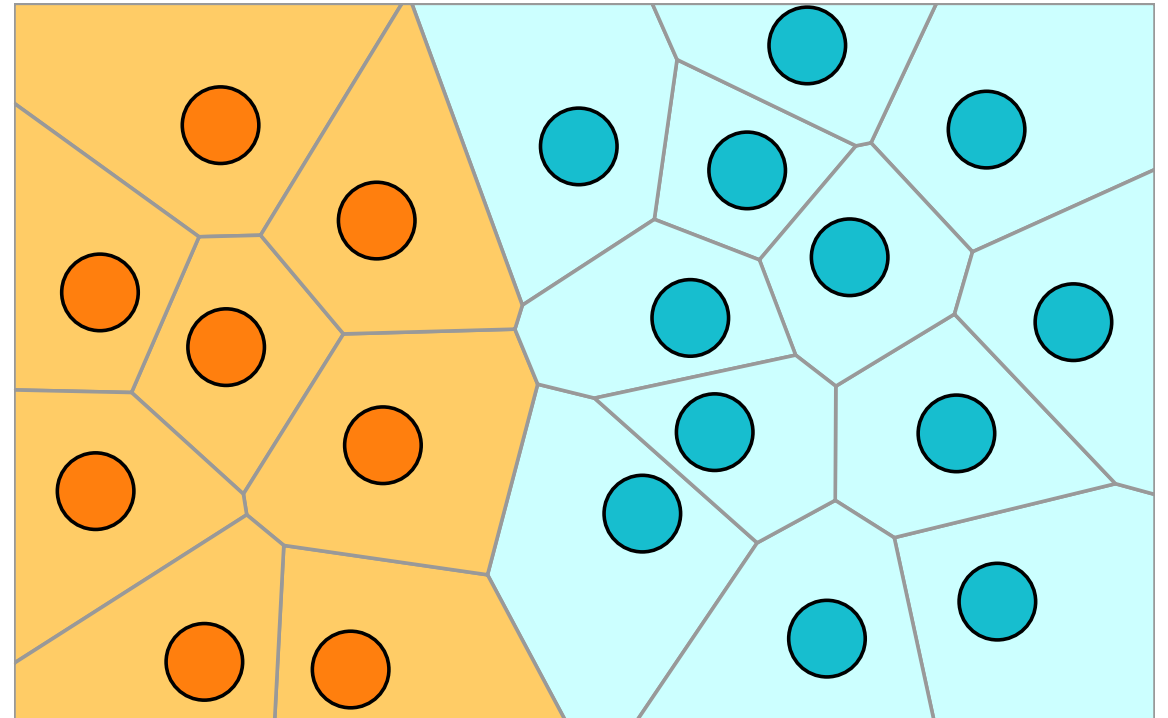
Linear classifiers



This image is CC0 1.0 public domain

# Linear Classification vs. Nearest Neighbors

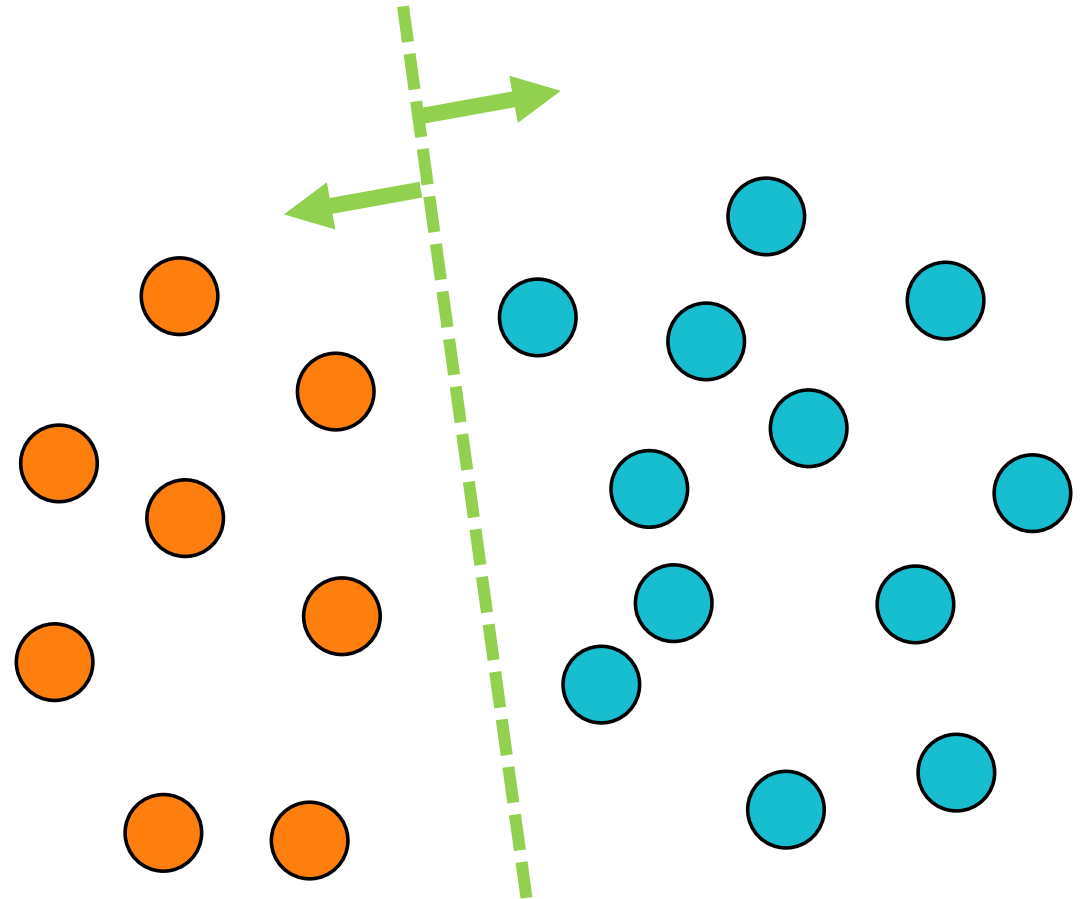
- Nearest Neighbors
  - Store every image
  - Find nearest neighbors at test time, and assign same class





# Linear Classification vs. Nearest Neighbors

- Nearest Neighbors
  - Store every image
  - Find nearest neighbors at test time, and assign same class
- Linear Classifier
  - Store hyperplanes that best separate different classes
  - We can compute continuous class score by calculating distance from hyperplane



We can interpret this as a linear "score function" for each class.

# Score functions



class scores

# Parametric Approach

Parametric approach



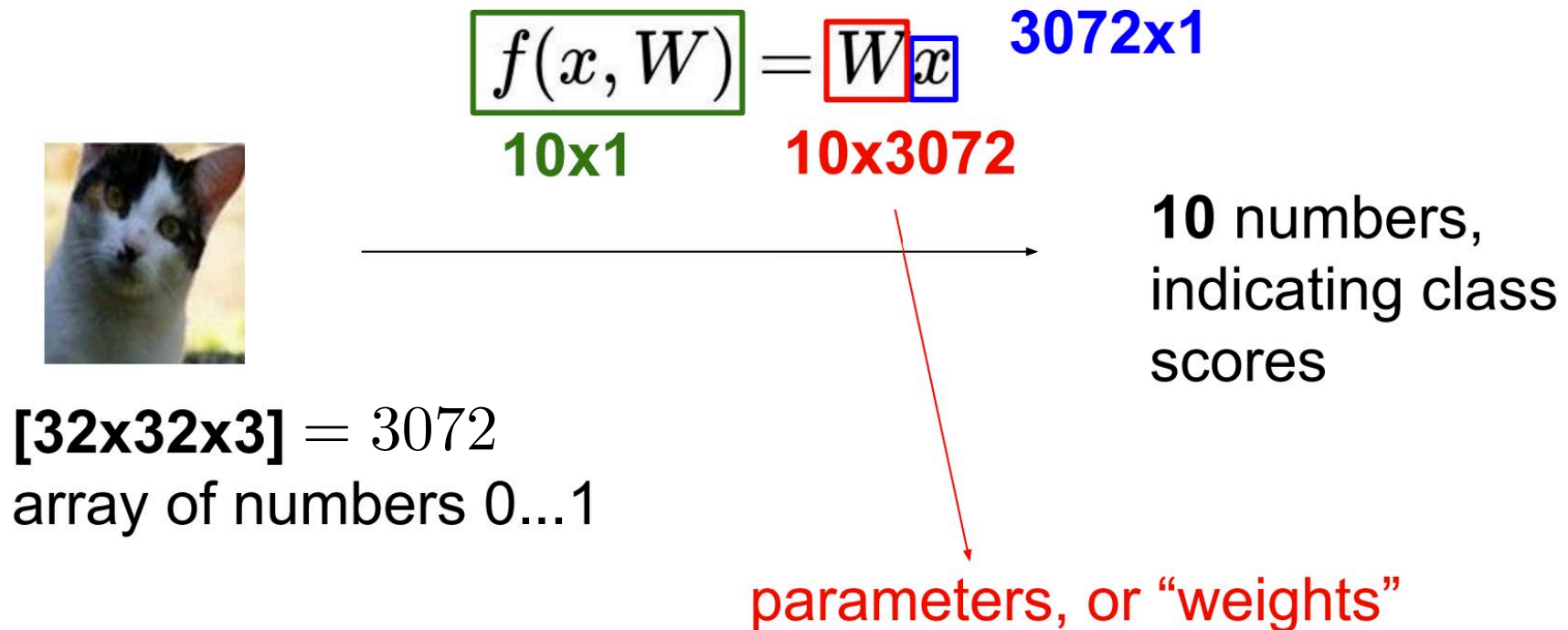
image parameters

$f(\mathbf{x}, \mathbf{W})$

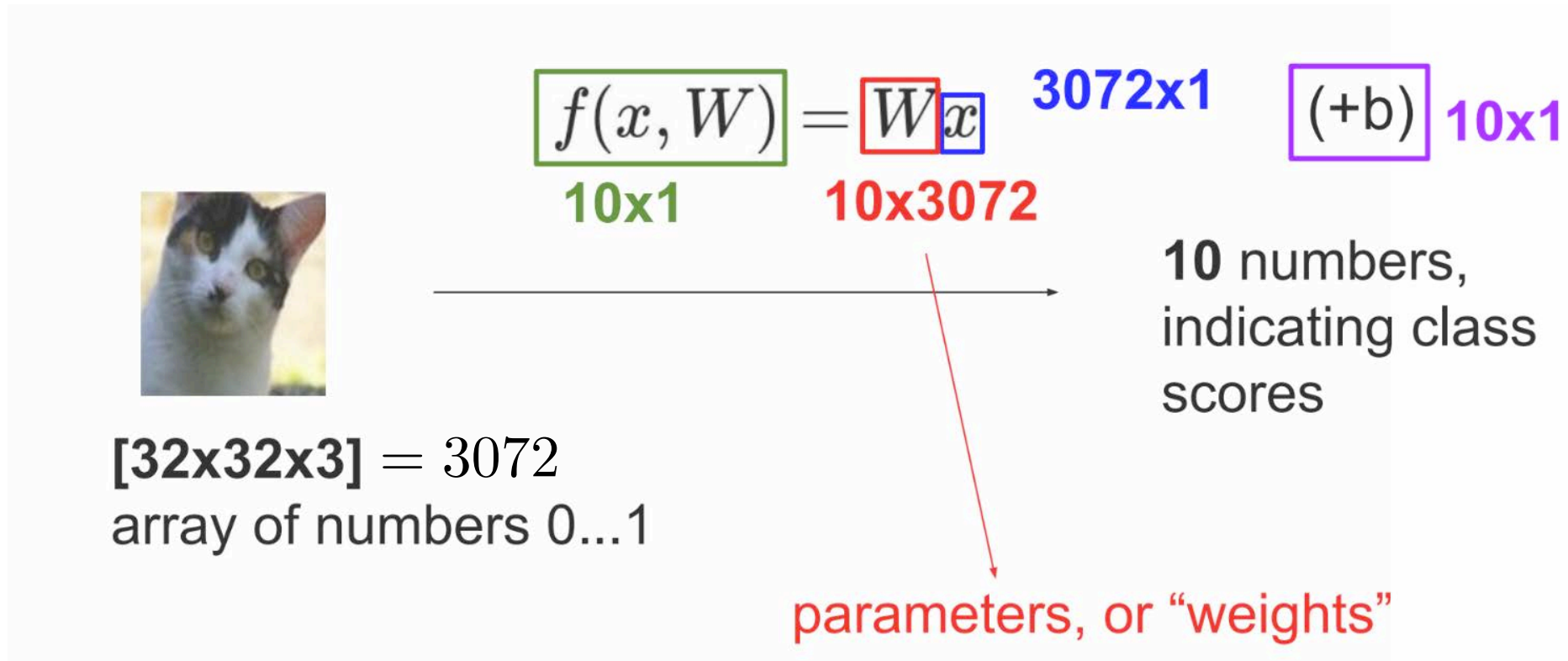
**10** numbers,  
indicating class  
scores

**[32x32x3]** = 3072  
array of numbers 0...1  
(3072 numbers total)

# Parametric Approach: Linear Classifier



# Parametric Approach: Linear Classifier



# Linear Classifier

define a **score function**

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

The diagram shows the equation  $f(x_i, W, b) = Wx_i + b$  with several annotations. An arrow points from the text "data (image)" to the variable  $x_i$ . Another arrow points from the text "weights" to the variable  $W$ . A third arrow points from the text "bias vector" to the variable  $b$ . A fourth arrow points from the text "parameters" to the entire right-hand side of the equation,  $Wx_i + b$ . A fifth arrow points from the text "class scores" to the function  $f$  on the left-hand side of the equation.

data (image)

class scores

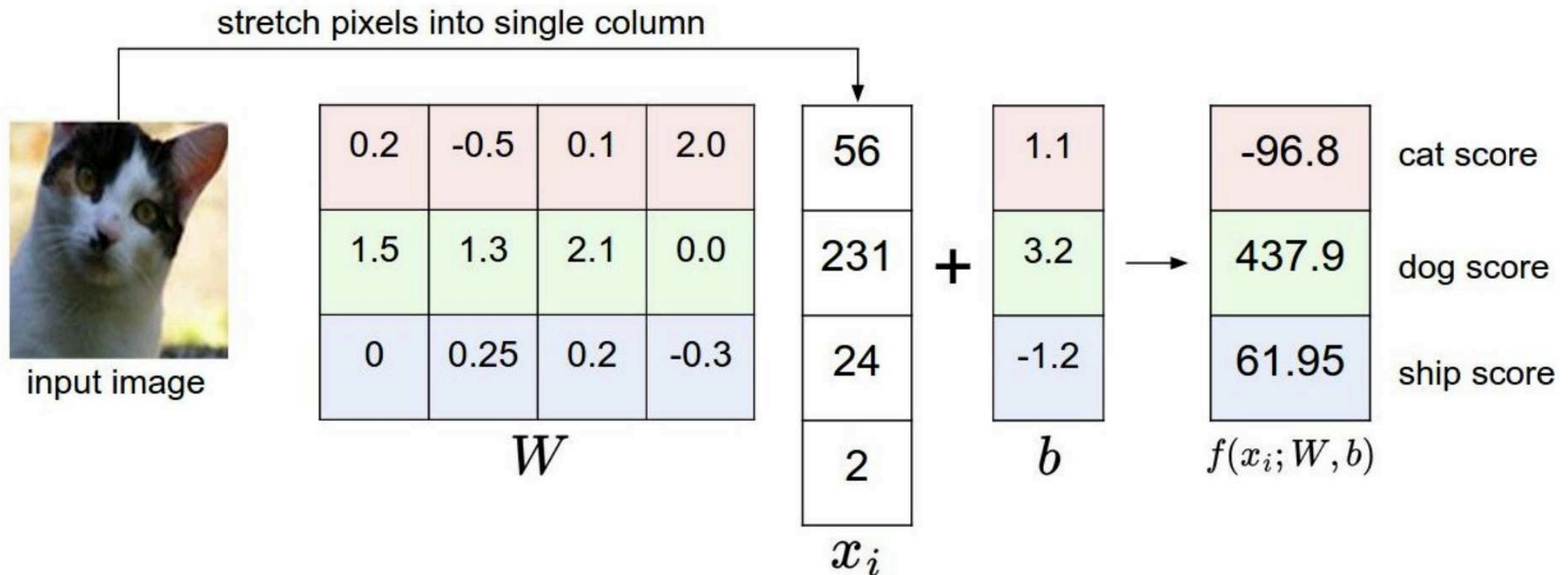
"weights"

"bias vector"

"parameters"

# Interpretation: Algebraic

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



# Interpretation: Geometric

- Parameters define a hyperplane for each class:

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

- We can think of each class score as defining a distribution that is proportional to distance from the corresponding hyperplane





# Interpretation: Template matching

- We can think of the rows in  $W$  as templates for each class



Rows of  $W$  in  $f(x_i, W, b) = Wx_i + b$

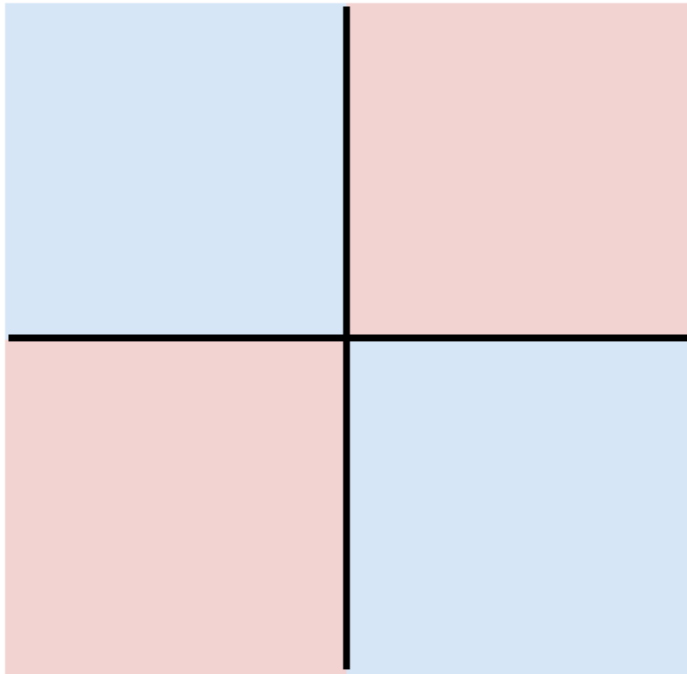
# Hard Cases for a Linear Classifier

**Class 1:**

First and third quadrants

**Class 2:**

Second and fourth quadrants

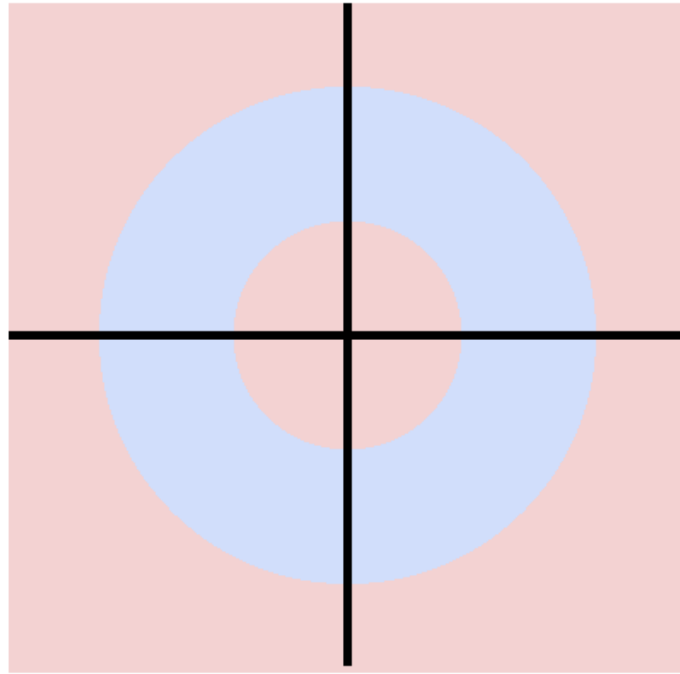


**Class 1:**

$1 \leq \text{L2 norm} \leq 2$

**Class 2:**

Everything else

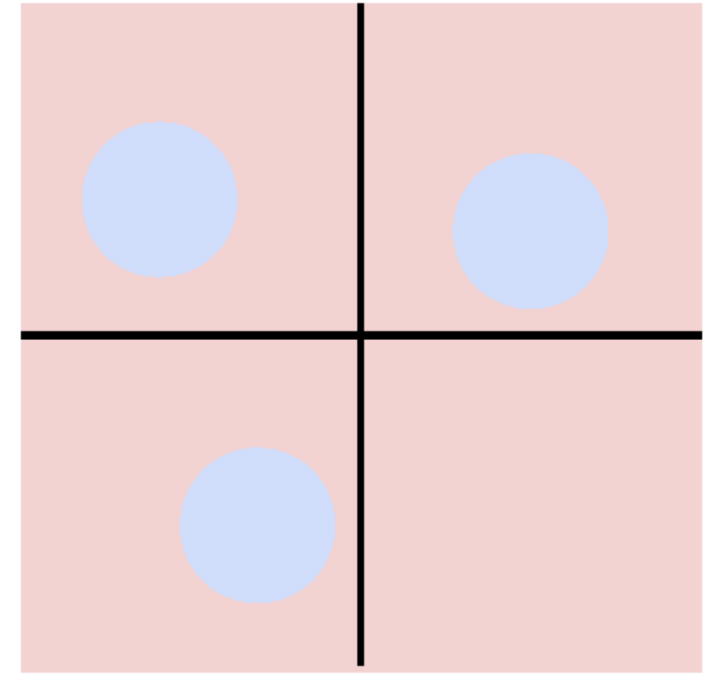


**Class 1:**

Three modes

**Class 2:**

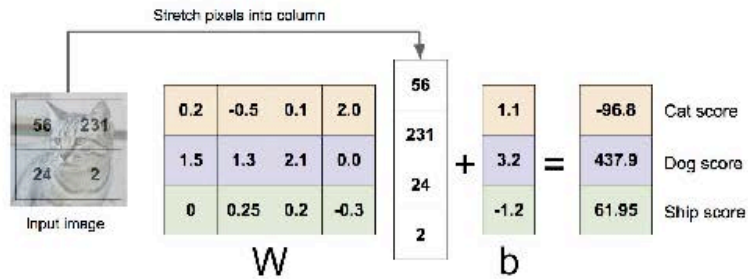
Everything else



# Linear Classifier: Three Viewpoints

## Algebraic Viewpoint

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



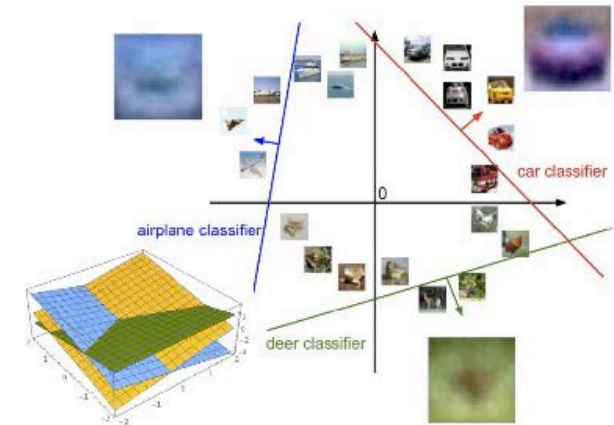
## Visual Viewpoint

One template  
per class



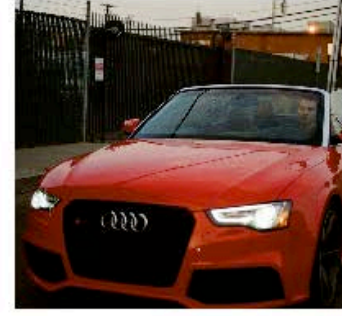
## Geometric Viewpoint

Hyperplanes  
cutting up space



**So far:** Defined a (linear) score function  $f(x, W) = Wx + b$

Example class scores for 3 images for some  $W$ :



How can we tell whether this  $W$  is good or bad?

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

# Recap

- Learning methods
  - k-Nearest Neighbors
  - **Linear classification**
- Classifier outputs a **score function** giving a score to each class
- How do we define how good a classifier is based on the training data?  
(Spoiler: define a *loss function*)

# Linear classification



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

Cat image by Nikita is licensed under CC-BY 2.0; Car image is CC0 1.0 public domain; Frog image is in the public domain

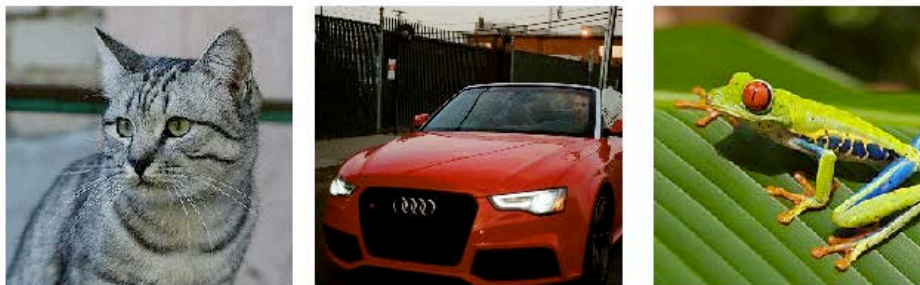
Output scores

TODO:

1. Define a **loss function** that quantifies our unhappiness with the scores across the training data.
2. Come up with a way of efficiently finding the parameters that minimize the loss function.  
**(optimization)**

# Loss Functions

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



|      |            |            |             |
|------|------------|------------|-------------|
| cat  | <b>3.2</b> | 1.3        | 2.2         |
| car  | 5.1        | <b>4.9</b> | 2.5         |
| frog | -1.7       | 2.0        | <b>-3.1</b> |

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 sum of loss over examples:

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

# Loss function, cost/objective function

- Given ground truth labels ( $y_i$ ), scores  $f(x_i, \mathbf{W})$ 
  - how unhappy are we with the scores?
- Loss function or objective/cost function measures unhappiness
- During training, **want to find the parameters  $\mathbf{W}$  that minimizes the loss function**



# Simpler example: binary classification

- Two classes (e.g., “cat” and “not cat”)
  - AKA “positive” and “negative” classes



cat

not cat

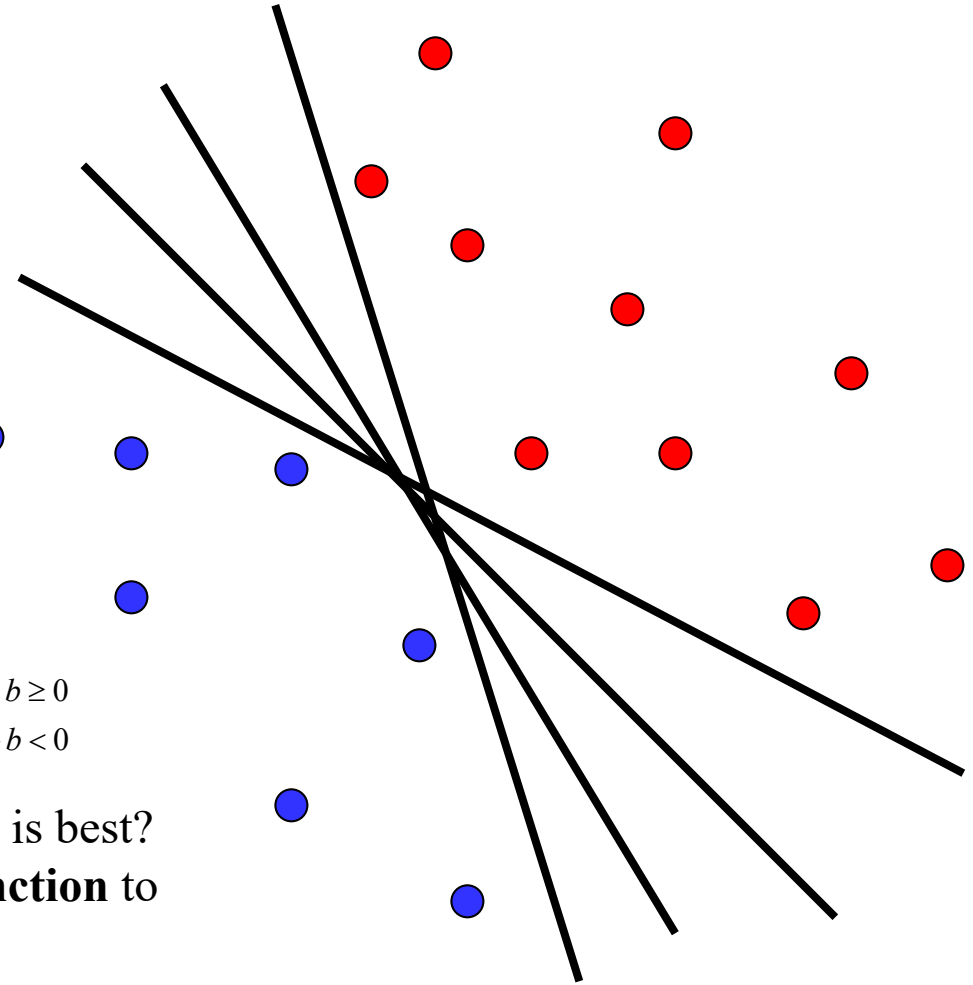
# Linear classifiers

- Find linear function (*hyperplane*) to separate positive and negative examples

$$x_i \text{ positive: } x_i \cdot w + b \geq 0$$

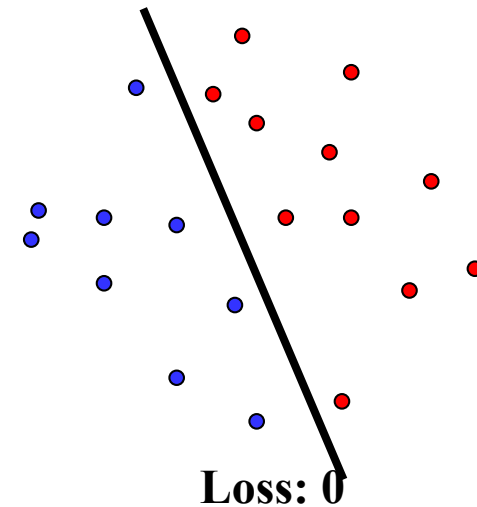
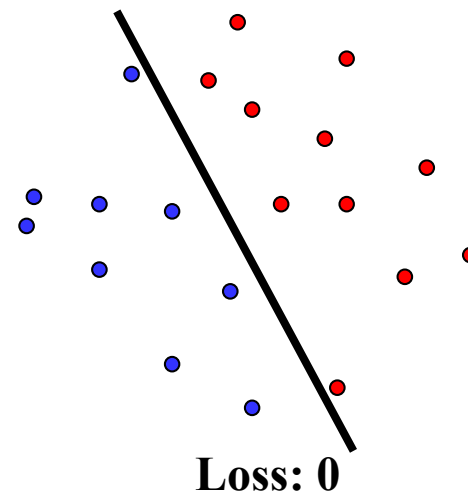
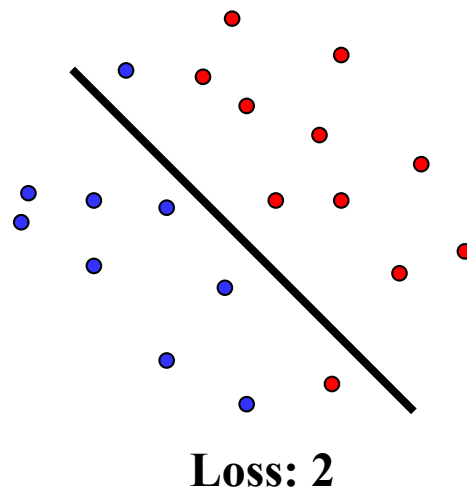
$$x_i \text{ negative: } x_i \cdot w + b < 0$$

Which hyperplane is best?  
We need a **loss function** to  
decide



# What is a good loss function?

- One possibility: Number of misclassified examples
  - Problems: discrete, can't break ties
  - We want the loss to lead to *good generalization*
  - We want the loss to work for more than 2 classes



# Softmax classifier

- Interpret Scores as unnormalized log probabilities of classes

$$f(x_i, W) = Wx_i \quad (\text{score function})$$

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

softmax function

Squashes values into *probabilities* ranging from 0 to 1

$$P(y_i | x_i; W)$$

Example with three classes:

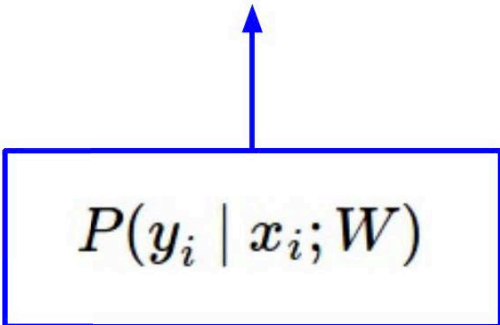
$$[1, -2, 0] \rightarrow [e^1, e^{-2}, e^0] = [2.71, 0.14, 1] \rightarrow [0.7, 0.04, 0.26]$$

# Cross-entropy loss

$$f(x_i, W) = Wx_i \quad (\text{score function})$$

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

$$L_i = -f_{y_i} + \log \sum_j e^{f_j}$$


$$P(y_i | x_i; W)$$

i.e. we're minimizing  
the negative log  
likelihood.

# Losses

- Cross-entropy loss is just one possible loss function
  - One nice property is that it reinterprets scores as probabilities, which have a natural meaning
- SVM (max-margin) loss functions also used to be popular
  - But currently, cross-entropy is the most common classification loss

# Summary

- Have score function and loss function
  - Currently, score function is based on linear classifier
  - Next, will generalize to convolutional neural networks
- Find  $W$  and  $b$  to minimize loss

$$L = \frac{1}{N} \sum_i -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2$$