# CS 4803 / 7643: Deep Learning

Topics:

- Image Classification
- Supervised Learning view
- K-NN

Dhruv Batra

Georgia Tech

# Administrativia

- PS0
  - Due: Aug 20 11:59pm
- More seats
  - We were able to recruit 1 more TA
  - 25 more seats added to 7643
- Piazza
  - 117/~200 people signed up. Please use that for questions.
- Office hours start next week
- ~~Gradescope/Canvas~~
  - Anybody not have access?
  - Please post on Piazza

# What is the collaboration policy?

- Collaboration
  - Only on HWs and project (not allowed in HW0).
  - You may discuss the questions
  - Each student writes their own answers
  - Write on your homework anyone with whom you collaborate
  - Each student must write their own code for the programming part

- Zero tolerance on plagiarism
  - Neither ethical nor in your best interest
  - Always credit your sources
  - Don’t cheat. We will find out.

# Python+Numpy Tutorial

CS231n Convolutional Neural Networks for Visual Recognition

## Python Numpy Tutorial

This tutorial was contributed by Justin Johnson.

We will use the Python programming language for all assignments in this course. Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

We expect that many of you will have some experience with Python and numpy; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

http://cs231n.github.io/python-numpy-tutorial/

# Plan for Today

- Image Classification
- Supervised Learning view
- K-NN

- Next time: Linear Classifiers

# Image Classification

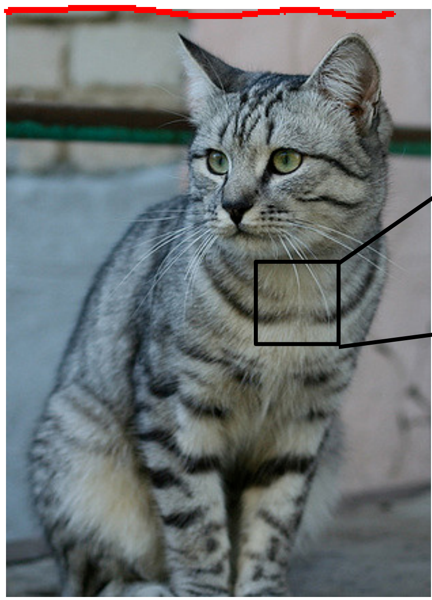
# **Image Classification**: A core task in Computer Vision



(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

→ cat

# **The Problem**: Semantic Gap



This image by Nikita is licensed under CC-BY 2.0

3 W

H

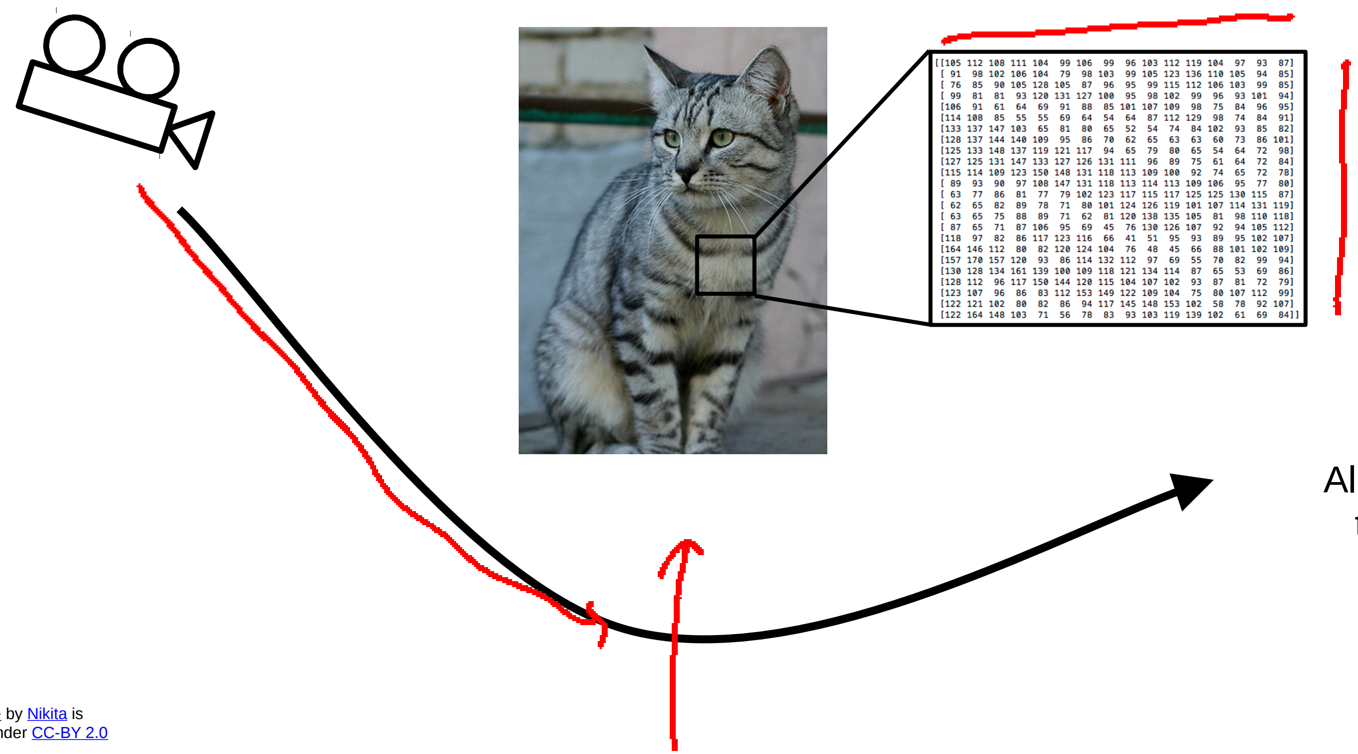
```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97 [illegible]  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105 [illegible] [illegible]]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99 [illegible]]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84 [illegible]]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)

# **Challenges**: Viewpoint variation

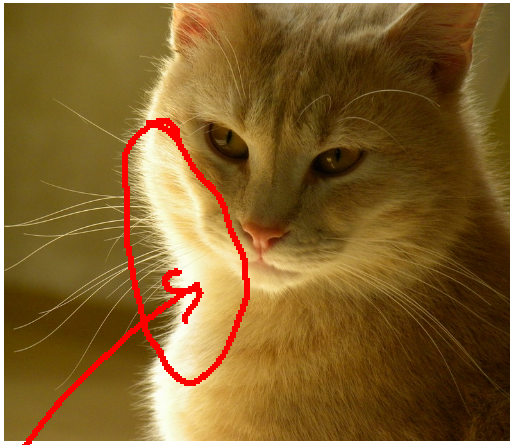



All pixels change when the camera moves!

# **Challenges**: Illumination

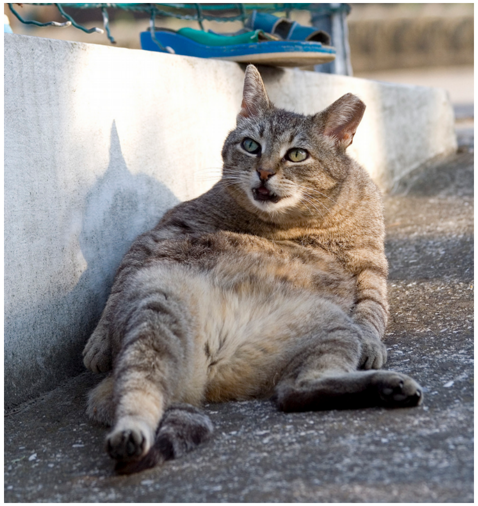








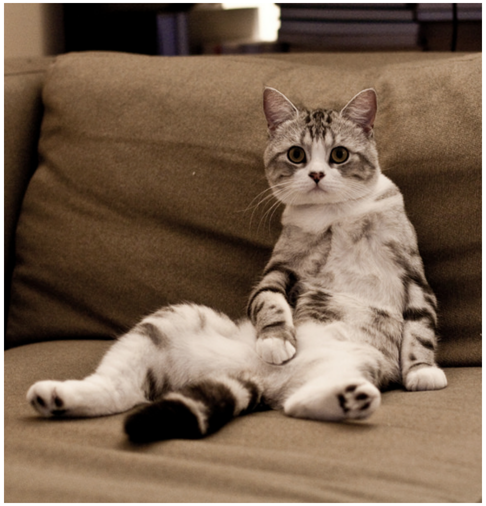
# **Challenges**: Deformation



This image by Umberto Salvagnin is licensed under CC-BY 2.0

This image by Umberto Salvagnin is licensed under CC-BY 2.0

This image by sare bear is licensed under CC-BY 2.0



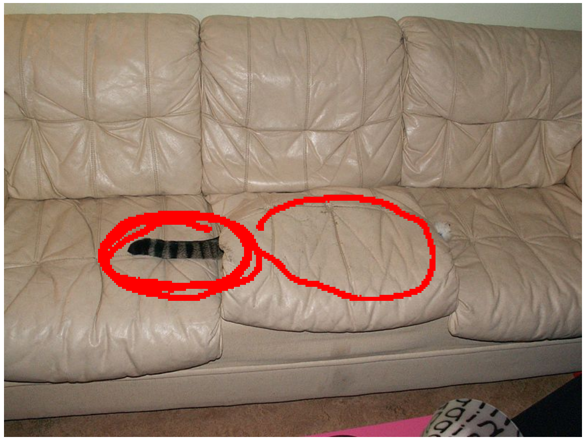
This image by Tom Thai is licensed under CC-BY 2.0

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

# **Challenges**: Occlusion







# **Challenges**: Background Clutter

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

# An image classifier

```
def classify_image(image):
  # Some magic here?
  return class_label
```

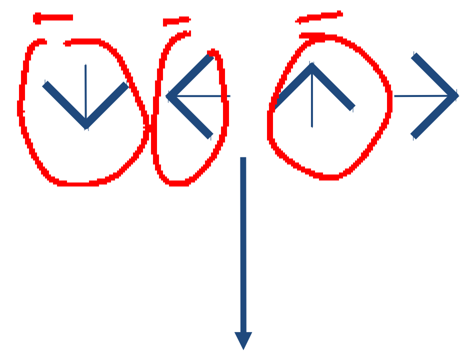
Unlike e.g. sorting a list of numbers,

**no obvious way** to hard-code the algorithm for recognizing a cat, or other classes.

# Attempts have been made

Find edges

Find corners



?

John Canny, "A Computational Approach to Edge Detection", IEEE TPAMI 1986

# ML: A Data-Driven Approach

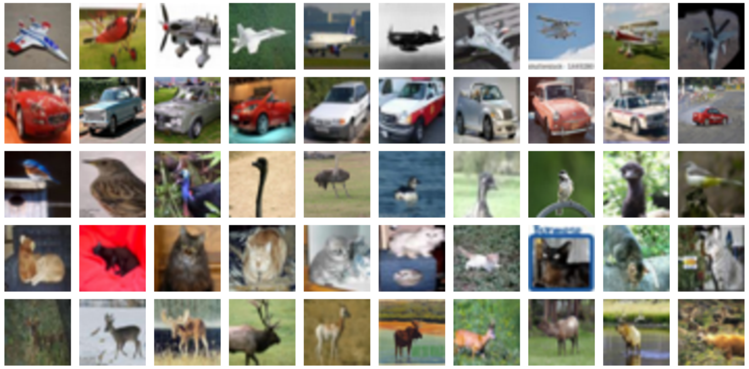
1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

```python
def train(images, labels):
  # Machine learning!
  return model
```

```python
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

**Example training set**

airplane
automobile
bird
cat
deer



# Notation

Scalars: $x, y, z \in \mathbb{R}^1$

Vectors: $\vec{x}, \vec{y} \in \mathbb{R}^n$

Matrices, $X, Y$
RVs, Sets

Input Dimension: $d$ $\quad \vec{x} \in \mathbb{R}^d$

Output / # classes $k$ $\quad \vec{y} \in \mathbb{R}^k$

# samples $n, N$

parameters: $\vec{w}, \vec{\theta}$

# Supervised Learning

$\in \mathbb{R}^1 \quad \mathbb{R}^d \quad \mathbb{R}^{H \times W \times 3}$

- Input: x (images, text, emails…)

$\in \{-1, +1\}$

- Output: y (cats vs dogs, spam vs not…)

$\in \{1, \ldots, k\} \quad \in \mathbb{R}^k$

- (Unknown) Target Function
  - f: X ✉ Y (the "true" mapping / reality)
- Data set
  - { (x₁,y₁), (x₂,y₂), …, (x_N,y_N) } $= \mathcal{D}$

$f(x_1)$

Goal: find f

Predict $f(\vec{x})$ at new $\vec{x}$

# Supervised Learning

① Model Class / Hypothesis Set

$$\mathcal{H} = \{ h : X \to Y \} \qquad \vec{y} = h(\vec{x})$$

② Loss Function

$$Loss(h, D) = \frac{1}{N} \sum_{i=1}^{N} \boxed{L_i}\left( h(x_i), y_i \right)$$

③ Learning = Search / Opt in M.C.

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}}\ Loss(h, D)$$

# Supervised Learning

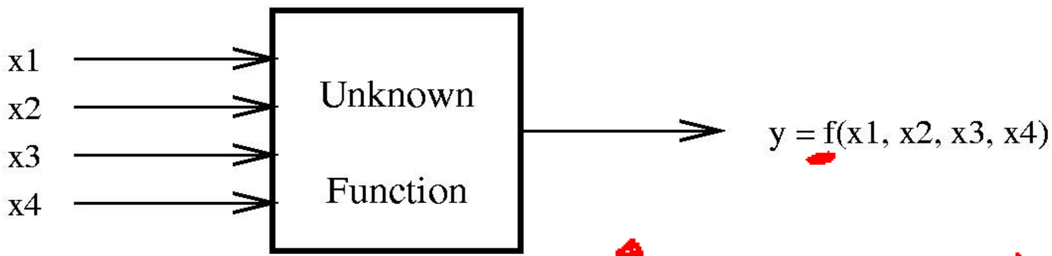
- Input: x (images, text, emails…)
- Output: y (spam or non-spam…)
- (Unknown) Target Function
  - f: X  Y (the "true" mapping / reality)
- Data
  - $\{ (x_1,y_1), (x_2,y_2), \ldots, (x_N,y_N) \}$
- Model / Hypothesis Class
  - H = {h: X  Y}
  - e.g. $y = h(x) = \text{sign}(w^T x)$
- Learning = Search in hypothesis space
  - Find best h in model class.

# Learning is hard!

$2^{[2^d - n]}$ = #fns compatible with dataset

## A Learning Problem

$\mathcal{H} = \{h : X \to Y\}$




#fn

d - boolean x's

1 boolean output

$2^{2^d} = |\mathcal{H}|$

$2^4 = 16$

| Example | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 1 | 0 | 0 |
| 6 | 1 | 1 | 0 | 0 | 0 |
| 7 | 0 | 1 | 0 | 1 | 0 |

n

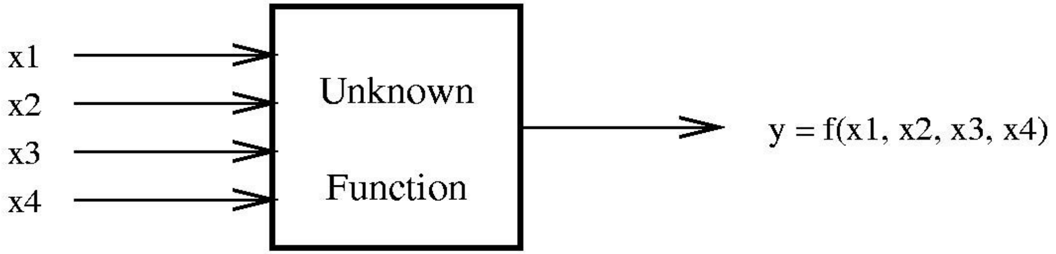
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | ? |

$f = x_4 \wedge (\bar{x}_2)$

# Learning is hard!

- No assumptions = No learning

**A Learning Problem**




| Example | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $y$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 1 | 0 | 0 |
| 6 | 1 | 1 | 0 | 0 | 0 |
| 7 | 0 | 1 | 0 | 1 | 0 |

# Procedural View

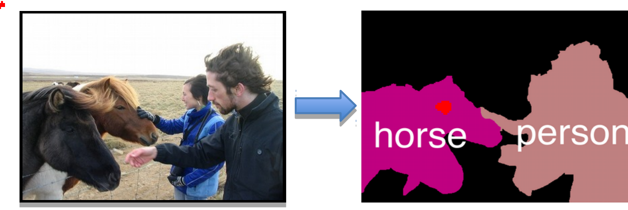
- Training Stage:
  - Training Data { ($x_i$,$y_i$) } → $\boxed{h}$ (Learning)
- Testing Stage
  - Test Data x → h(x) (Apply function, Evaluate error)

# Statistical Estimation View

- Probabilities to rescue:
  - X and Y are *random variables*
  - D = $(x_1,y_1), (x_2,y_2), \ldots, (x_N,y_N)$ ~ P(X,Y)
- IID: Independent Identically Distributed
  - Both training & testing data sampled IID from P(X,Y)
  - Learn on training set
  - Have some hope of *generalizing* to test set

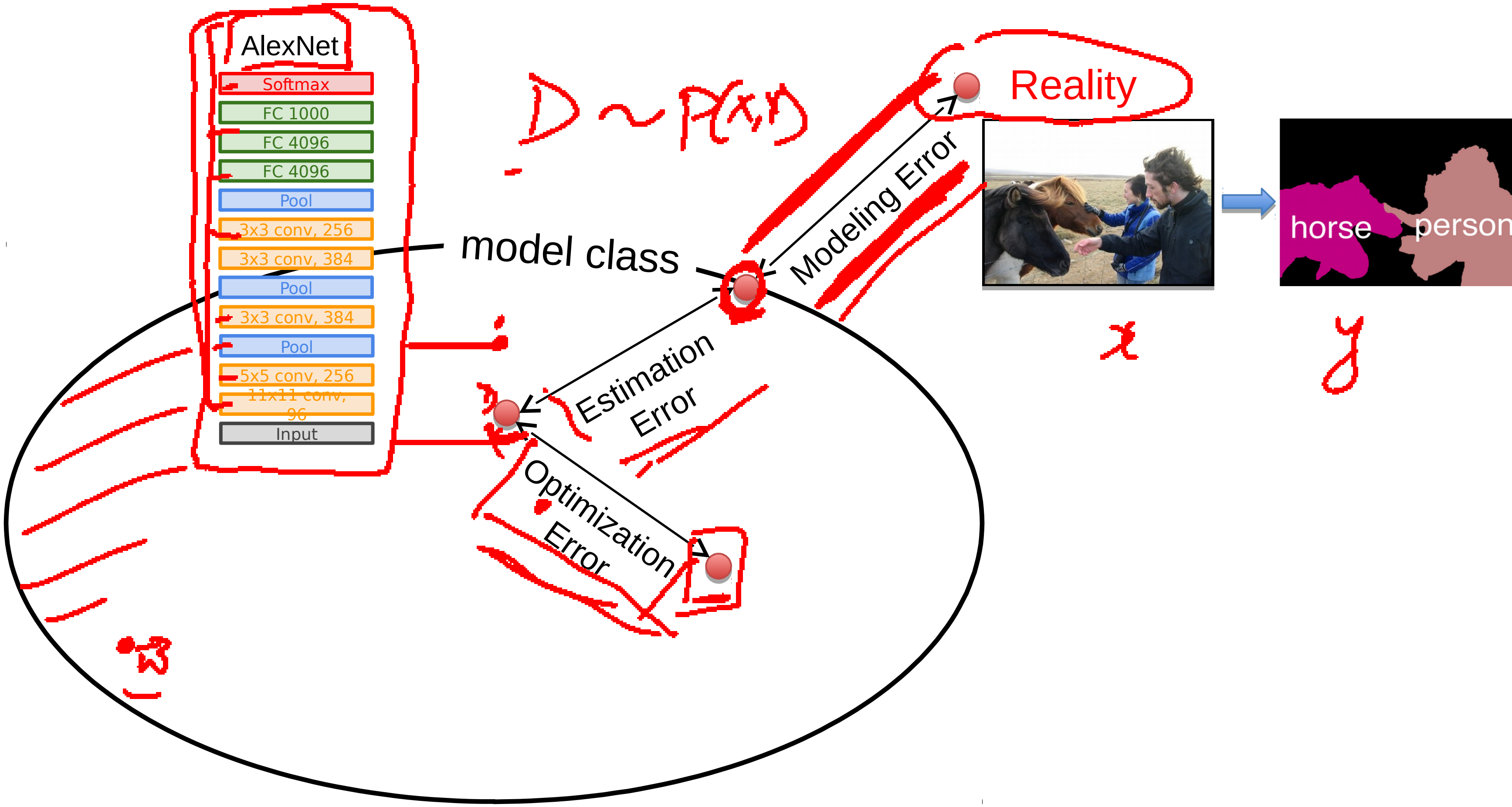
# Classical Learning Theory: Error Decomposition

- Reality

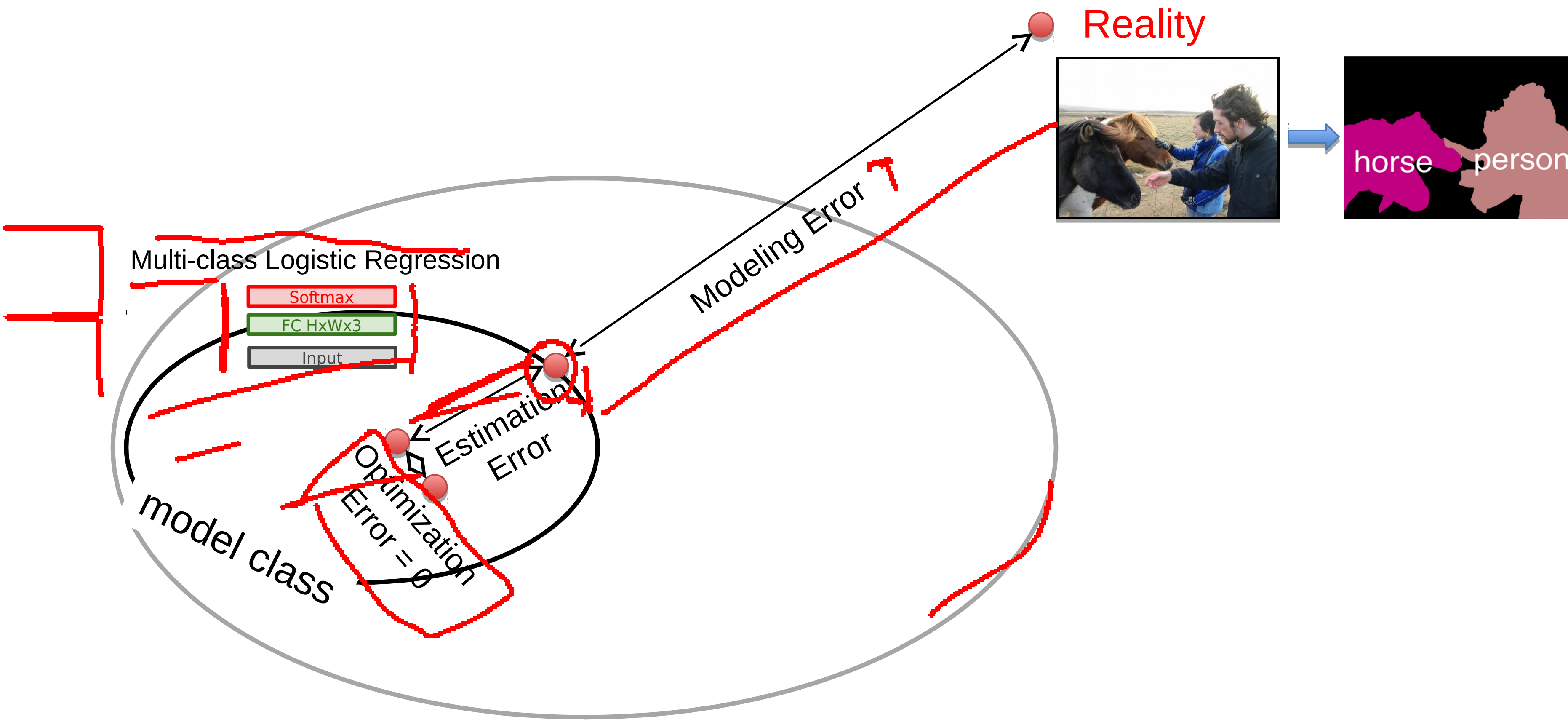



$x \rightarrow y$

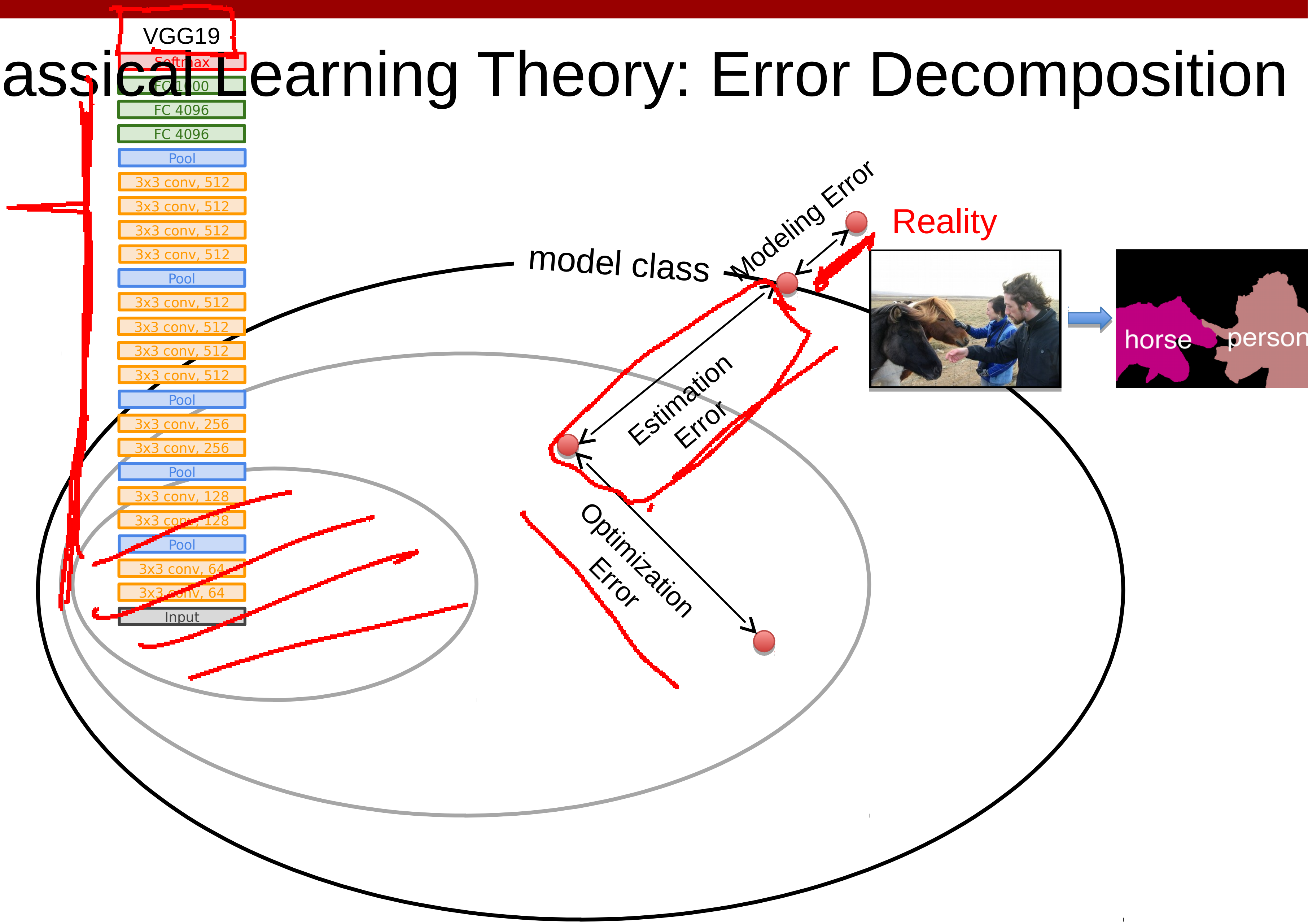
# Classical Learning Theory: Error Decomposition



# Classical Learning Theory: Error Decomposition



# Classical Learning Theory: Error Decomposition



# Error Decomposition

- Approximation/Modeling Error
  - You approximated reality with model
- Estimation Error
  - You tried to learn model with finite data
- Optimization Error
  - You were lazy and couldn't/didn't optimize to completion
- Bayes Error
  - Reality just sucks

# Caveats

- A number of recent empirical results question our intuitions built from this clean separation.

# First classifier: **Nearest Neighbor**

```python
def train(images, labels):
  # Machine learning!
  return model
```

→ Memorize all data and labels

```python
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

→ Predict the label of the most similar training image

# Example Dataset: **CIFAR10**

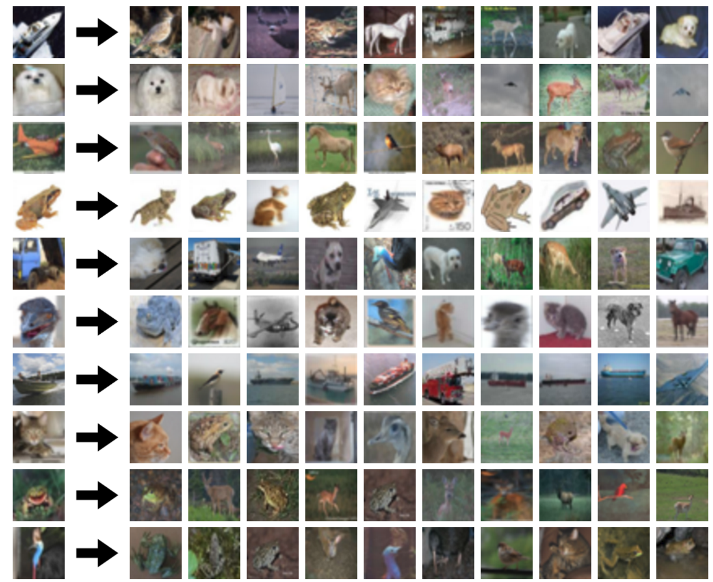
**10** classes
**50,000** training images
**10,000** testing images


Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009.

# Example Dataset: **CIFAR10**

**10** classes
**50,000** training images
**10,000** testing images

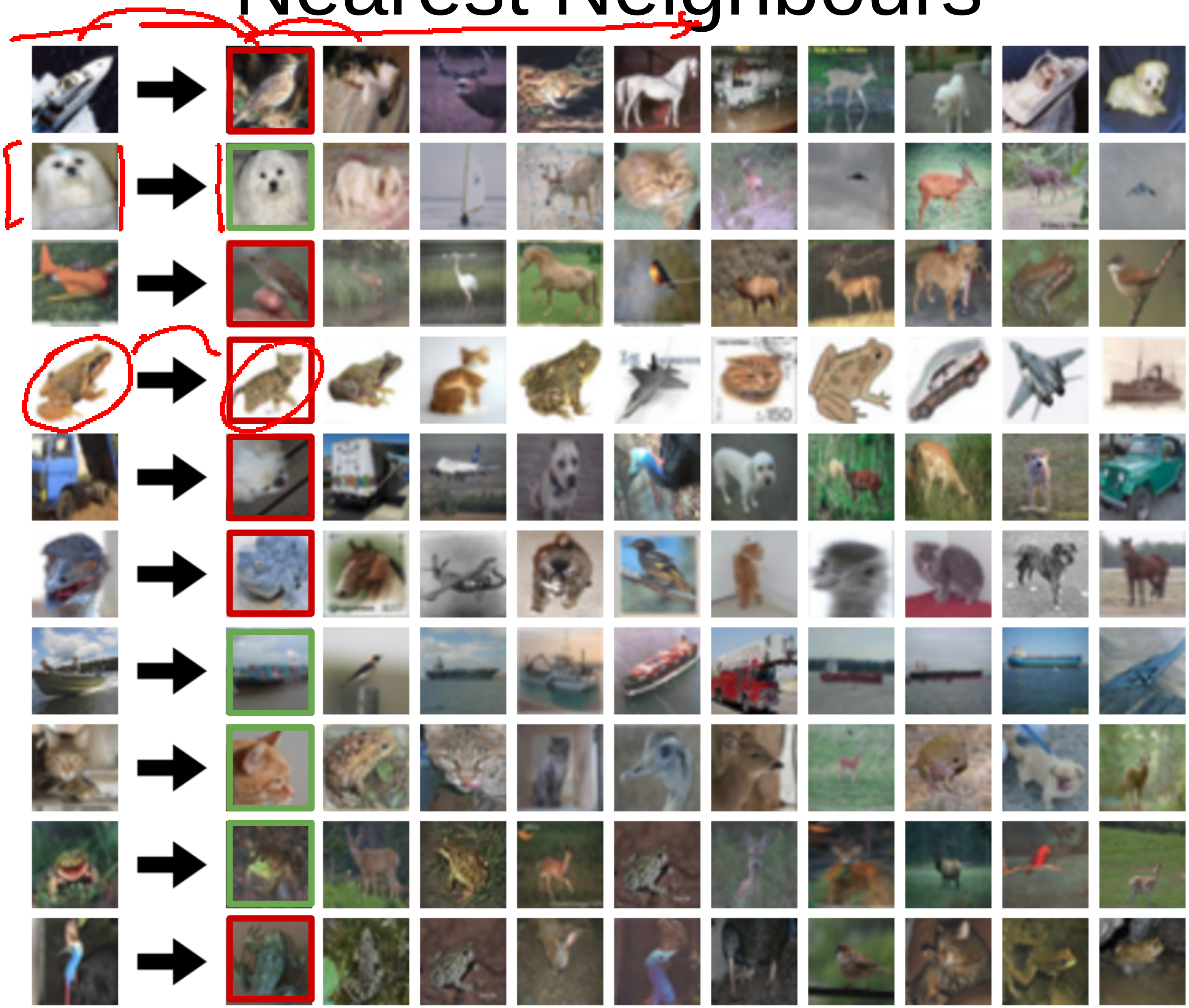

Test images and nearest neighbors



Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009.

# Nearest Neighbours

# Nearest Neighbours



# Instance/Memory-based Learning

Four things make a memory based learner:

- *A distance metric*
- *How many nearby neighbors to look at?*
- *A weighting function (optional)*
- *How to fit with the local points?*

# 1-Nearest Neighbour

Four things make a memory based learner:

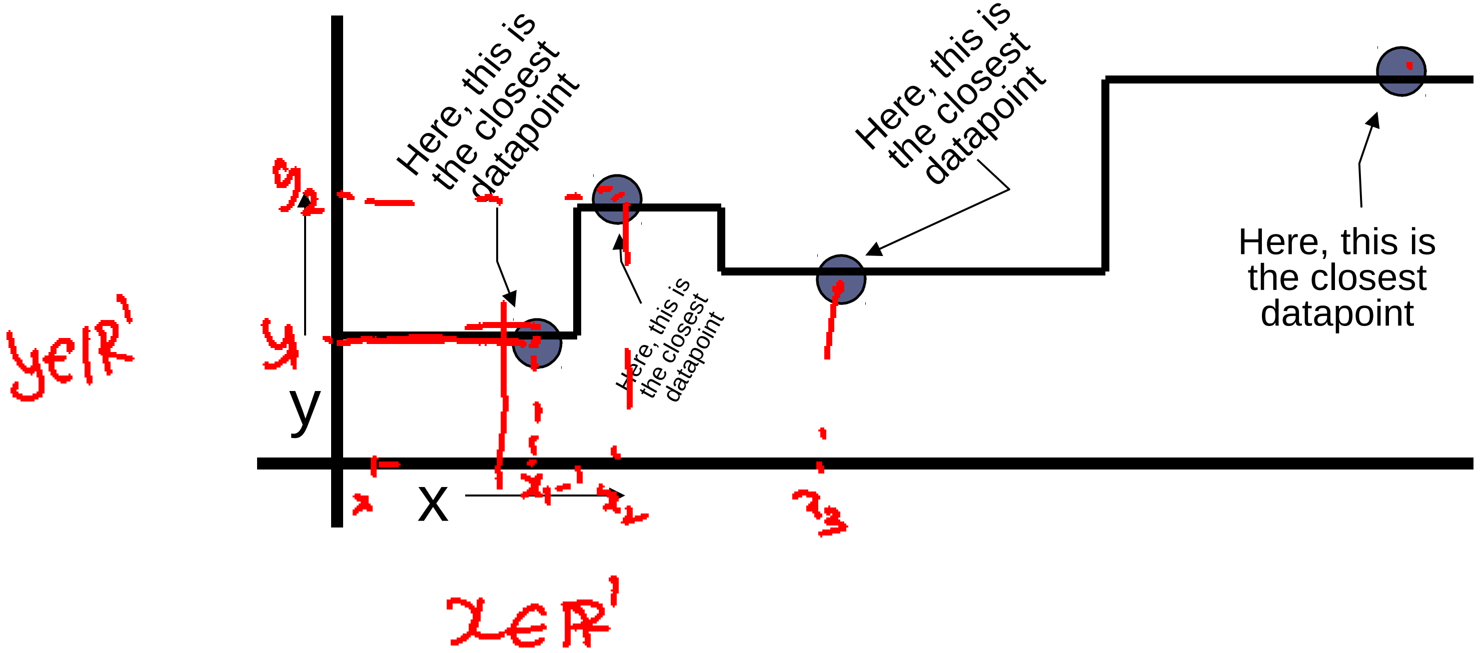
- *A distance metric*
  - **Euclidean (and others)**
- *How many nearby neighbors to look at?*
  - **1**
- *A weighting function (optional)*
  - **unused**
- *How to fit with the local points?*
  - **Just predict the same output as the nearest neighbour.**

# k-Nearest Neighbour

Four things make a memory based learner:

- *A distance metric*
  - **Euclidean (and others)**
- *How many nearby neighbors to look at?*
  - **k**
- *A weighting function (optional)*
  - **unused**
- *How to fit with the local points?*
  - **Just predict the average output among the nearest neighbours.**

# 1-NN for Regression



# **Distance Metric** to compare images

**L1 distance:** $$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

RGB

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

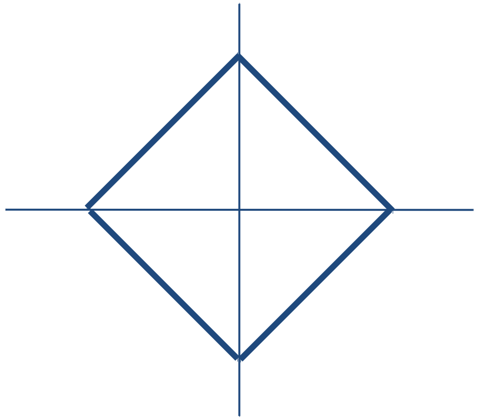
| | | | |
|---|---|---|---|
| 46 | 12 | 14 | 1 |
| 82 | 13 | 39 | 33 |
| 12 | 10 | 0 | 30 |
| 2 | 32 | 22 | 108 |

$\xrightarrow{\text{add}}$ 456

# K-Nearest Neighbors: Distance Metric

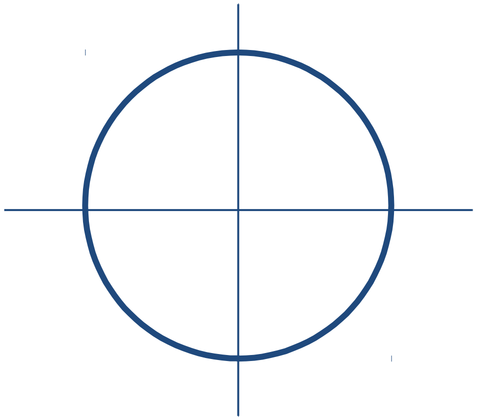
L1 (Manhattan) distance

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



L2 (Euclidean) distance

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



## Nearest Neighbor classifier

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Nearest Neighbor classifier

Memorize training data

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Nearest Neighbor classifier

For each test image:
  Find closest train image
  Predict label of nearest image

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

## Nearest Neighbor classifier

**Q:** With N examples, how fast are training and prediction?

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Nearest Neighbor classifier

**Q:** With N examples, how fast are training and prediction?

**A:** Train O(1), predict O(N)

```python
import numpy as np

class NearestNeighbor:
  def __init__(self):
    pass

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

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    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)

    # loop over all test rows
    for i in xrange(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 distance
      Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

    return Ypred
```

Nearest Neighbor classifier

**Q:** With N examples, how fast are training and prediction?

**A**: Train O(1),
predict O(N)

This is bad: we want classifiers that are **fast** at prediction; **slow** for training is ok

# Nearest Neighbour

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

# Hyperparameters

What is the best value of **k** to use?
What is the best **distance** to use?

These are **hyperparameters**: choices about the algorithm that we set rather than learn

# Hyperparameters

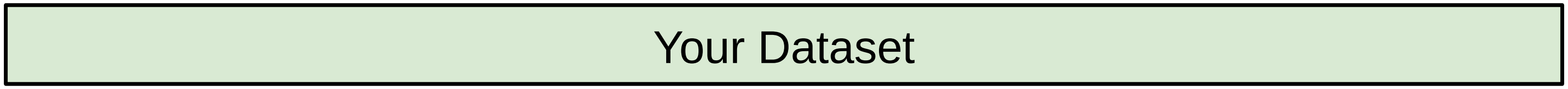
What is the best value of **k** to use?
What is the best **distance** to use?

These are **hyperparameters**: choices about the algorithm that we set rather than learn

Very problem-dependent.
Must try them all out and see what works best.

# Hyperparameters

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



# Hyperparameters

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

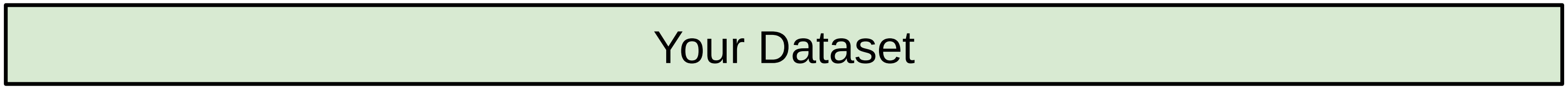
**BAD**: K = 1 always works perfectly on training data

Your Dataset

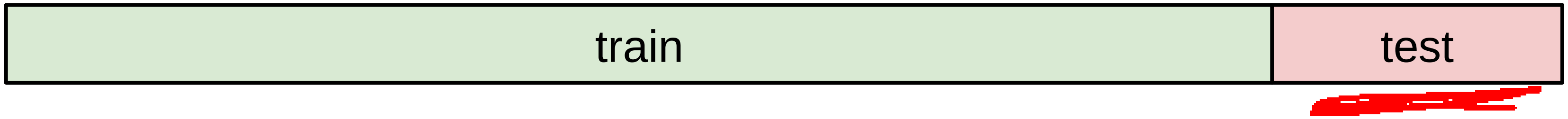
# Hyperparameters

**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



# Hyperparameters

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

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

Your Dataset

**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

train | test

# Hyperparameters

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

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

| Your Dataset |
| --- |

**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

| train | test |
| --- | --- |

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

**Better!**

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

$K^*, d^* = \max \; Acc(val)$

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

# Hyperparameters

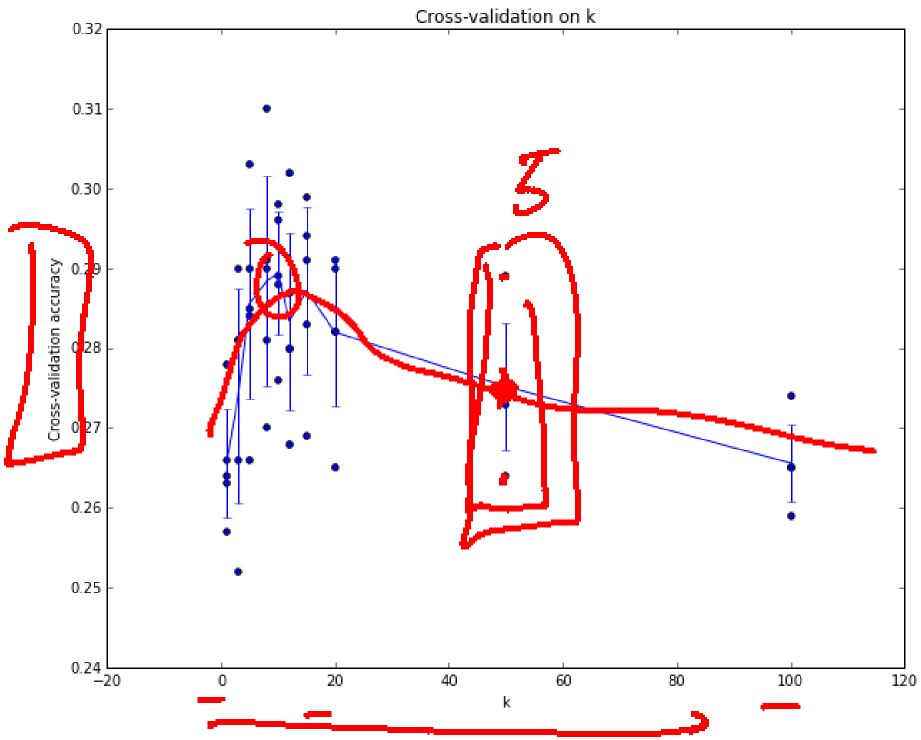
| Your Dataset |
|---|

**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

# Setting Hyperparameters




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 ~= 7 works best
for this data)

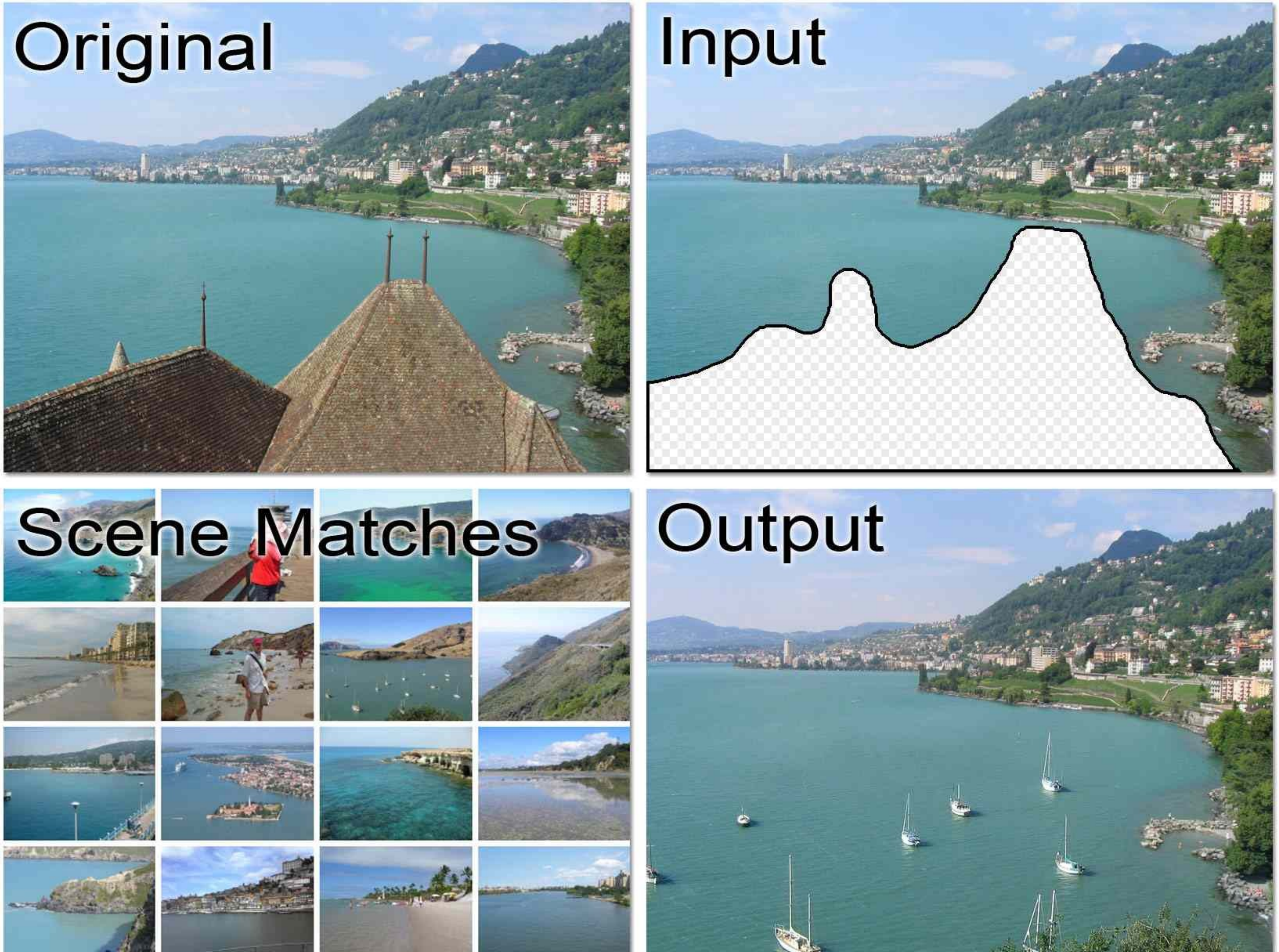
# Parametric vs Non-Parametric Models

- Does the capacity (size of hypothesis class) grow with size of training data?
  - Yes = Non-Parametric Models
  - No = Parametric Models

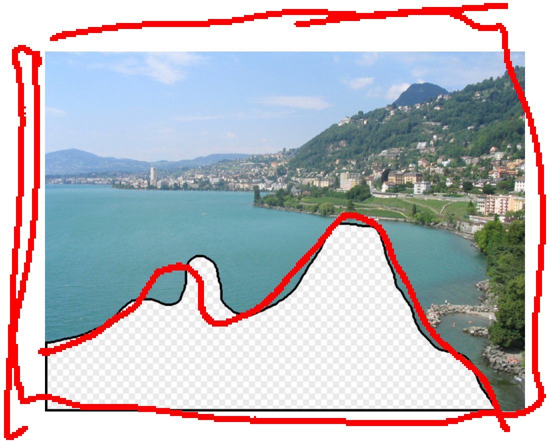
$h(\vec{x}) = \vec{w}^T \vec{x}$

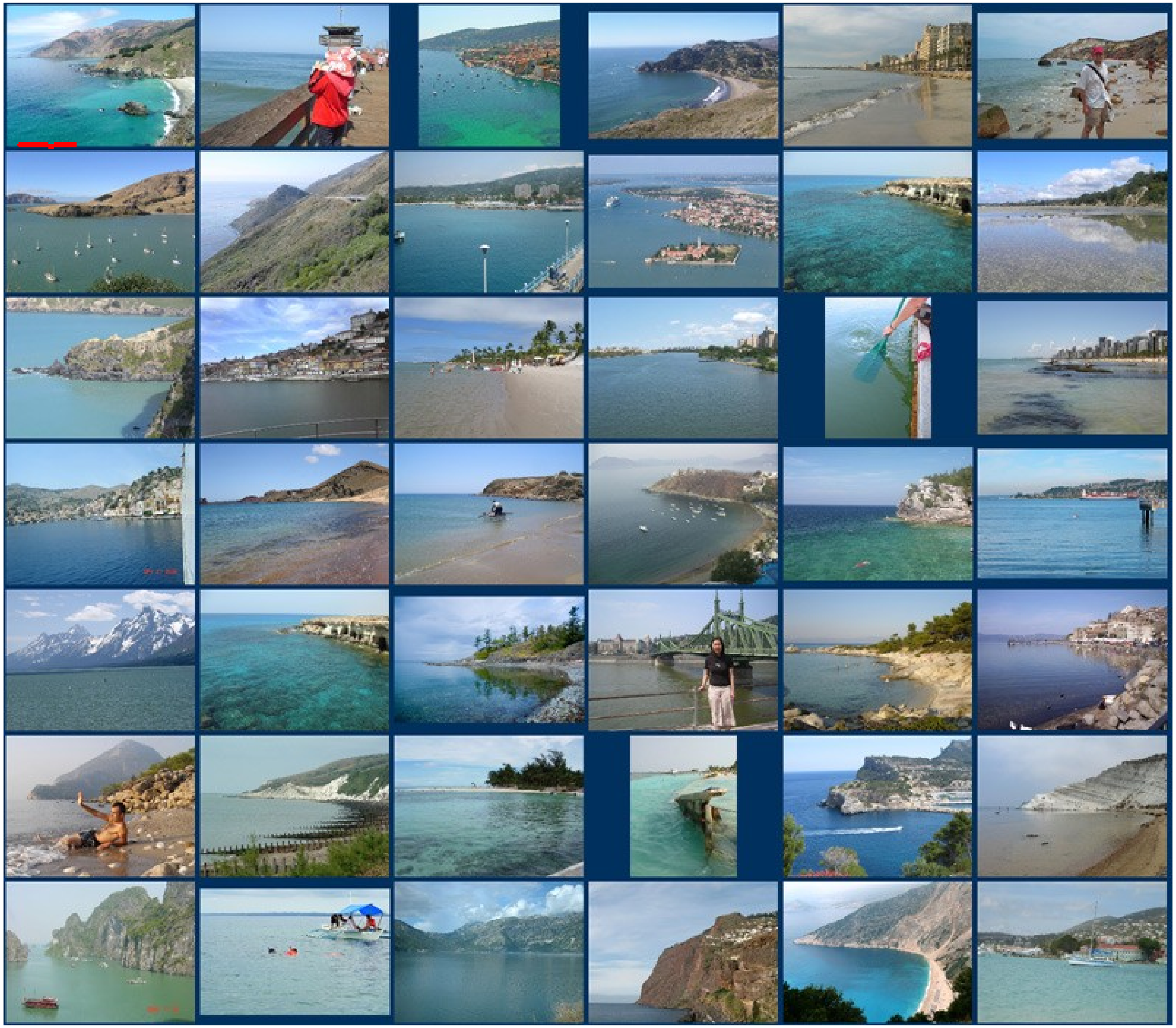
$h_{\vec{w}}$

# Scene Completion [Hayes & Efros, SIGGRAPH07]



Hays and Efros, SIGGRAPH 2007

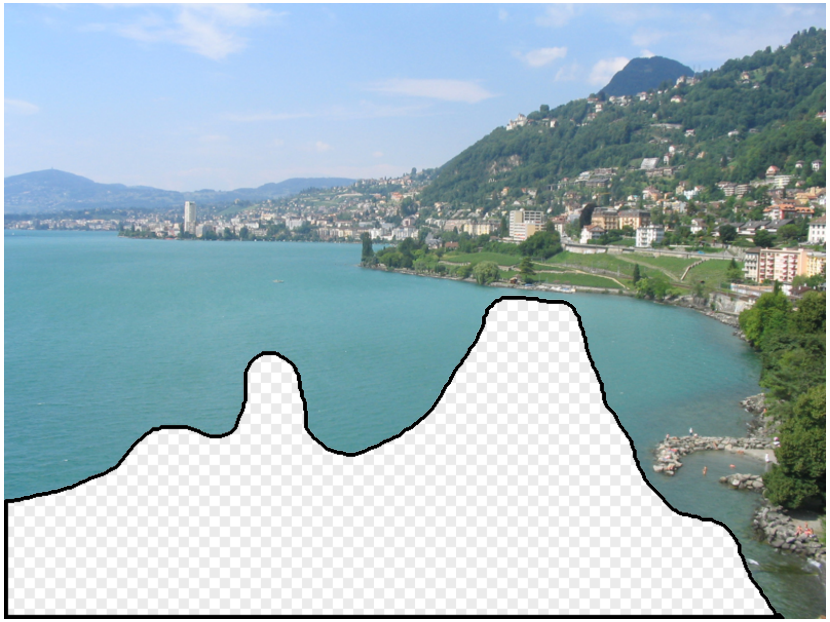




… 200 total

Hays and Efros, SIGGRAPH 2007

# Context Matching



Hays and Efros, SIGGRAPH 2007

Graph cut + Poisson blending

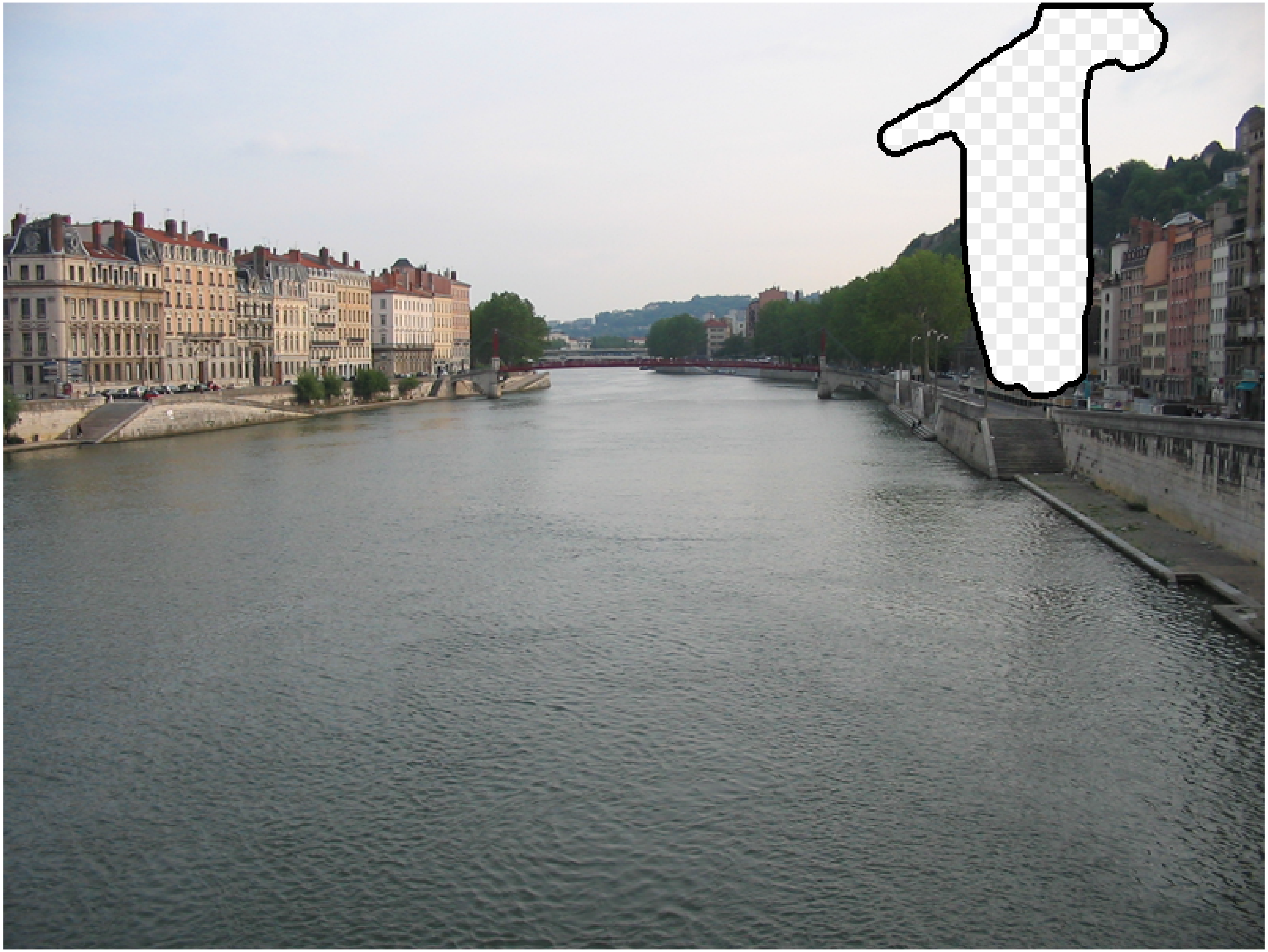
Hays and Efros, SIGGRAPH 2007

Hays and Efros, SIGGRAPH 2007


Hays and Efros, SIGGRAPH 2007


Hays and Efros, SIGGRAPH 2007

Hays and Efros, SIGGRAPH 2007



Hays and Efros, SIGGRAPH 2007

Hays and Efros, SIGGRAPH 2007

# Problems with Instance-Based Learning

- Expensive
  - No Learning: most real work done during testing
  - For every test sample, must search through all dataset – very slow!
  - Must use tricks like approximate nearest neighbour search

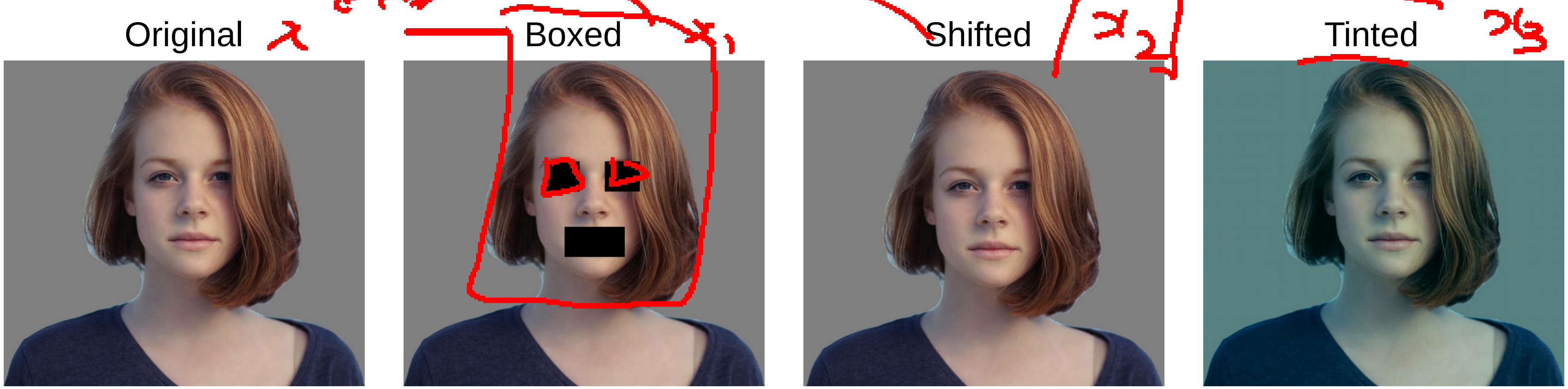
- Doesn't work well when large number of irrelevant features
  - Distances overwhelmed by noisy features

$[x_1, x_2$

- Curse of Dimensionality
  - Distances become meaningless in high dimensions
  - (See proof next)

# k-Nearest Neighbor on images **never used.**

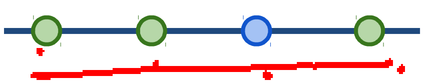
- Very slow at test time
- Distance metrics on pixels are not informative



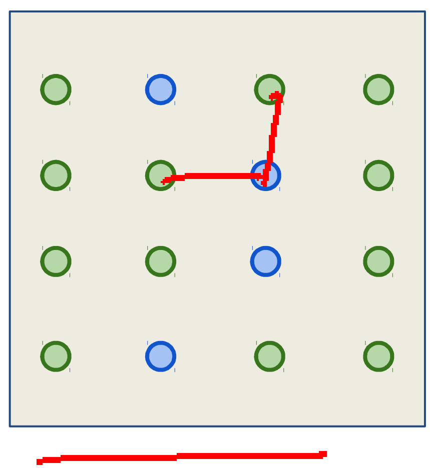

(all 3 images have same L2 distance to the one on the left) k-Nearest Neighbor on images **never used.**

- Curse of dimensionality

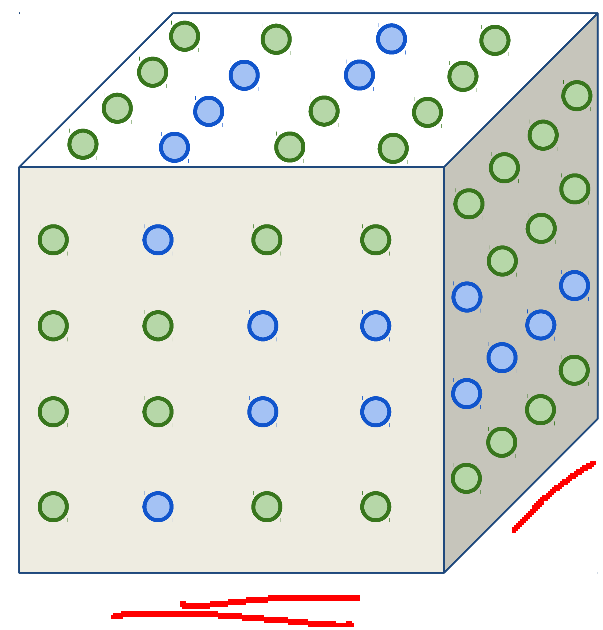
Dimensions = 1
Points = 4



Dimensions = 2
Points = $4^2$



Dimensions = 3
Points = $4^3$

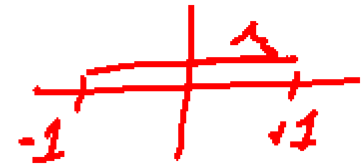


# Curse of Dimensionality

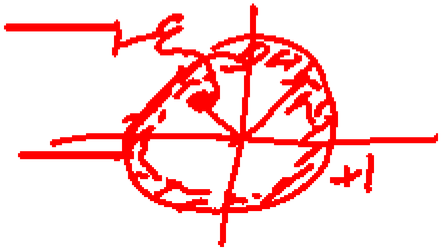
- Consider: Sphere of radius 1 in d-dims
- Consider: an outer ε-shell in this sphere
- What is $\frac{\textit{shell volume}}{\textit{sphere volume}}$ ?

Vol

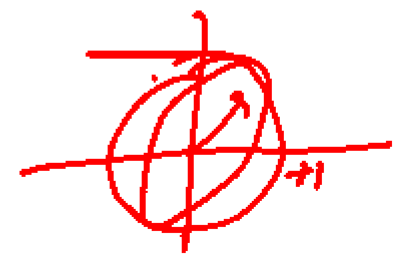
2r




$\pi r^2$




$\frac{4}{3}\pi r^3$




$$= \frac{k_d 1^d - k_d (1-\varepsilon)^d}{k_d 1^d} = 1-(\underbrace{1-\varepsilon}_{\to 0})^d$$

$\lim_{d\to\infty}$

$= 1$

$\vdots$

$k_d r^d$

d-dim

# Curse of Dimensionality

Ratio

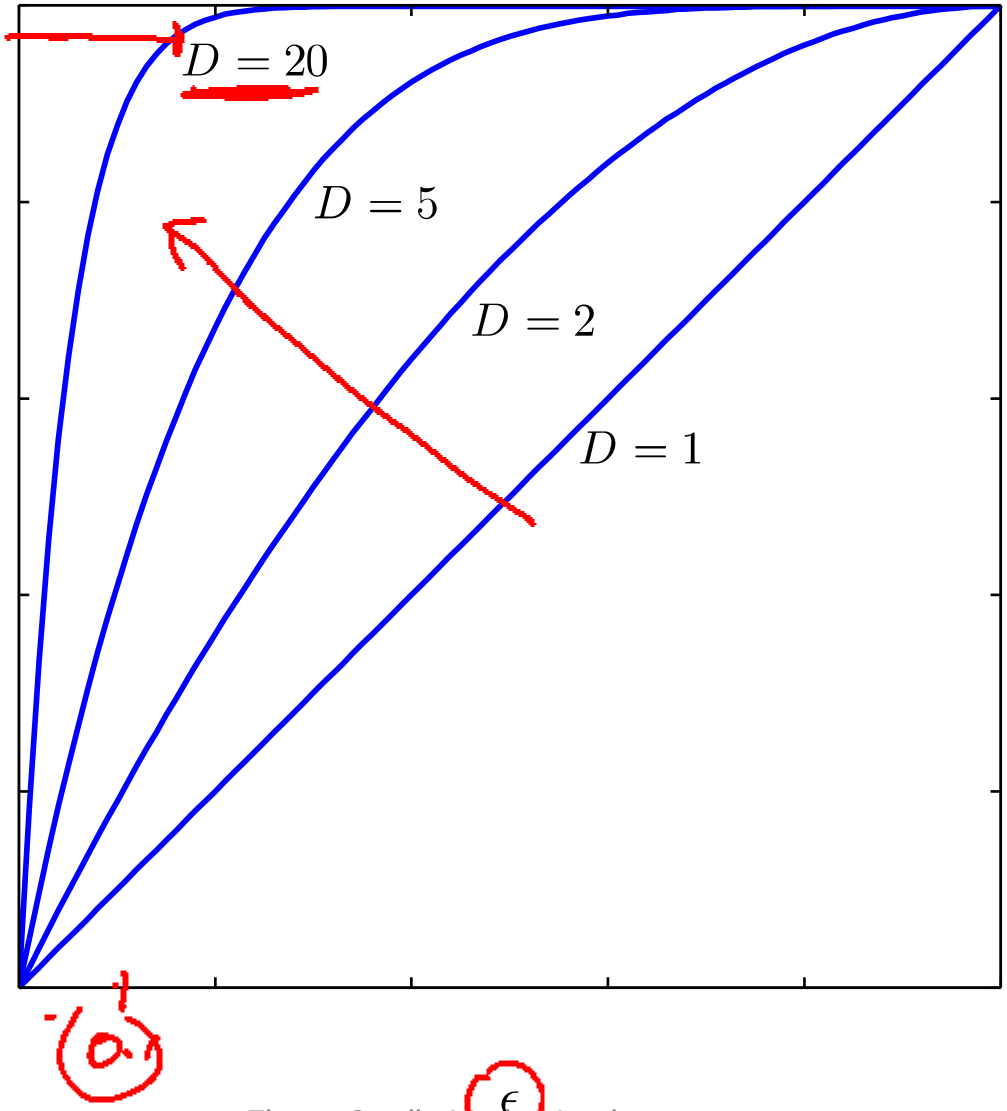





Figure Credit: Kevin Murphy

# 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!