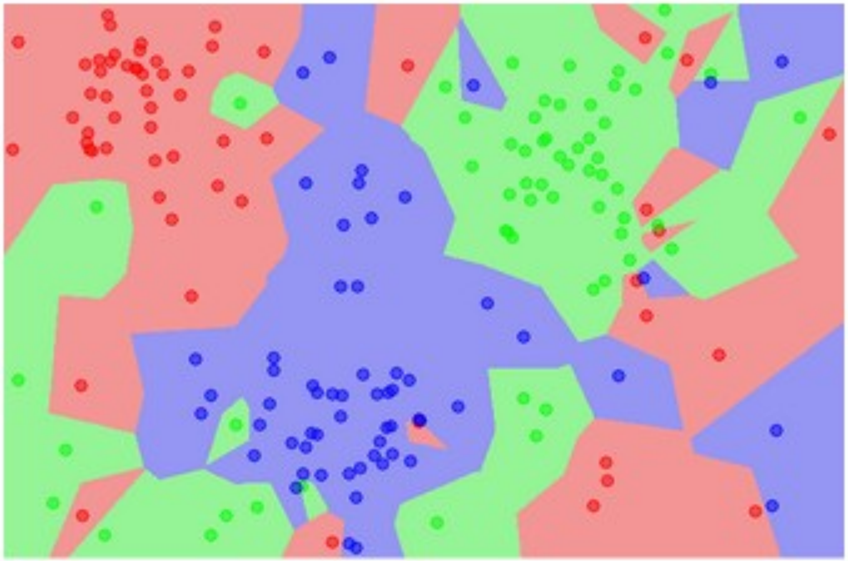
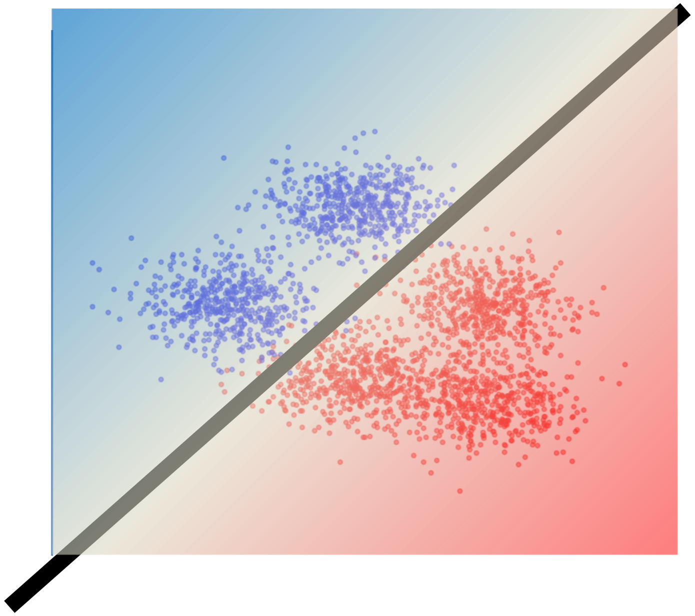
CSCI 497P/597P: Computer Vision

Scott Wehrwein

K Nearest Neighbor Classifier
Linear Classifiers





Reading

- http://cs231n.github.io/linear-classify/

Announcements

- No class tomorrow
- HW4 due Friday
- P3 due Monday

Goals

- Understand the standard ML pipeline for image classification problems:
  - Represent images as feature vectors
  - Learn a classifier function from labeled data
  - Classify novel images using the learned classifier
- Understand KNN classifier and why it doesn't work so well on images.
- Understand the importance of splitting data into train/val/test sets when developing algorithms and tuning hyperparameters.
- Understand the benefits and limitations of linear classifiers over KNN.
- Understand the mathematical formulation of a binary and multiclass linear classifier.

Image classification - Multilabel classification

Is this a dog? Yes
Is this furry? Yes
Is this sitting down? Yes

How are we going to solve this?

An image classifier

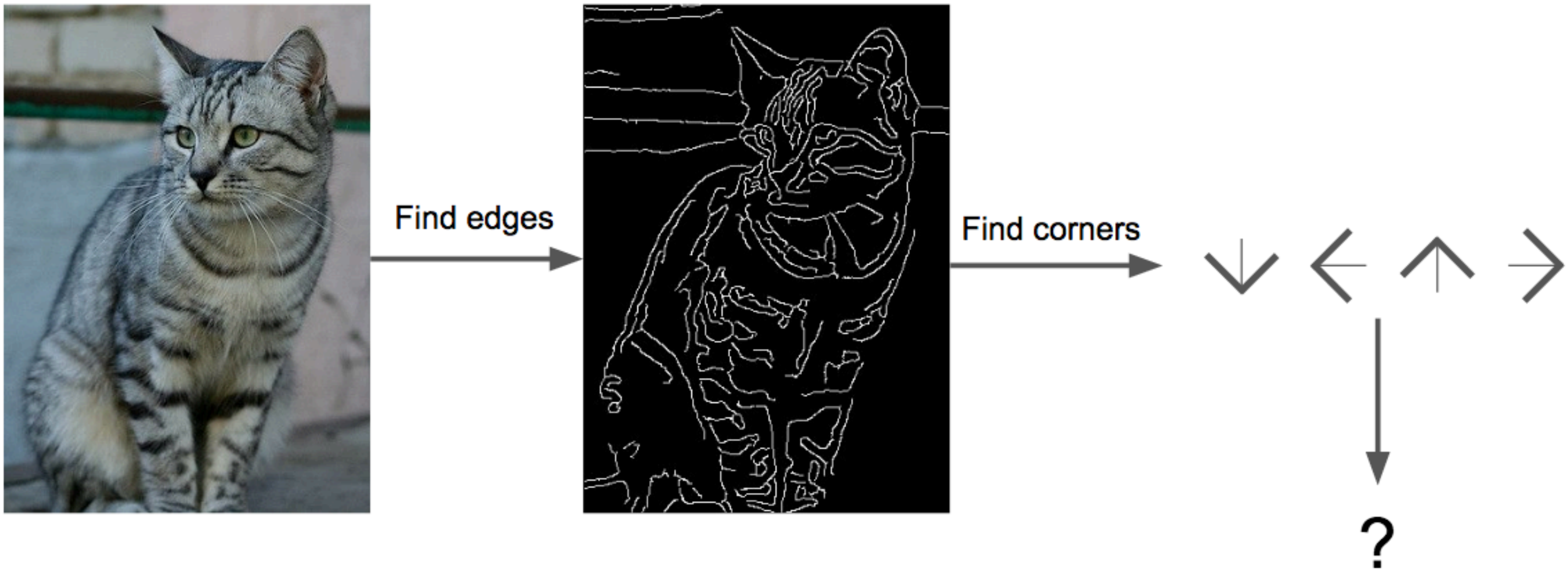
```
def classify_image(image):
  # Some magic here?
  return class_label
```

Unlike e.g. sorting a list of numbers,

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

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Attempts have been made



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

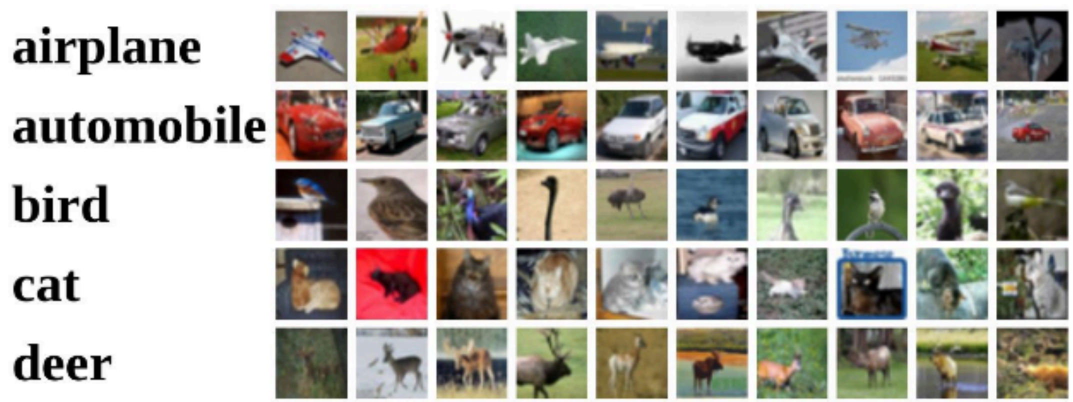
Machine Learning: Data-Driven Approach

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

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

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

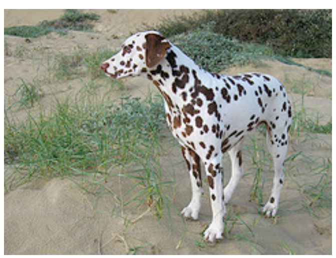
Example training set



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Representing Images

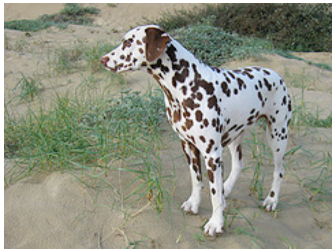
- We have images; ML works on vectors.
- To do machine learning, we need a function that takes an image and converts it into a vector.

ϕ () = 

- Given an image, use ϕ to get a vector representing a point in high dimensional space

Classifying Images

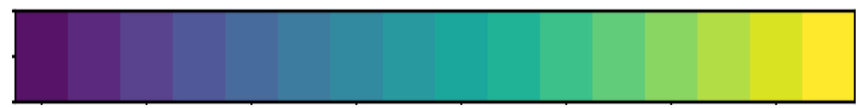
- Given an image, use ϕ to get a vector and plot it as a point in high dimensional space

ϕ () = 

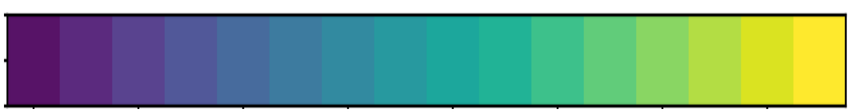
- Then, use a *classifier* function to map feature vectors to class labels:
- h() = “dog”

Classifying Images: Pipeline

1. Represent the image in some *feature space*

ϕ () = 

2. Classify the image based on its feature representation.

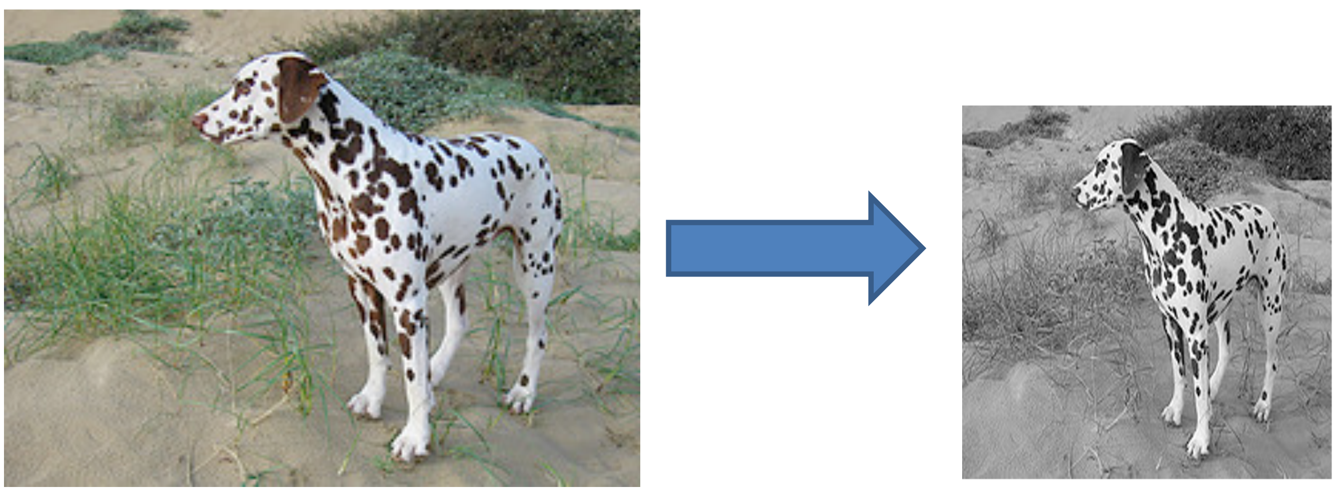
- h() = “dog”

Two important pieces

- The feature extractor (ϕ)
- The classifier (*h*)

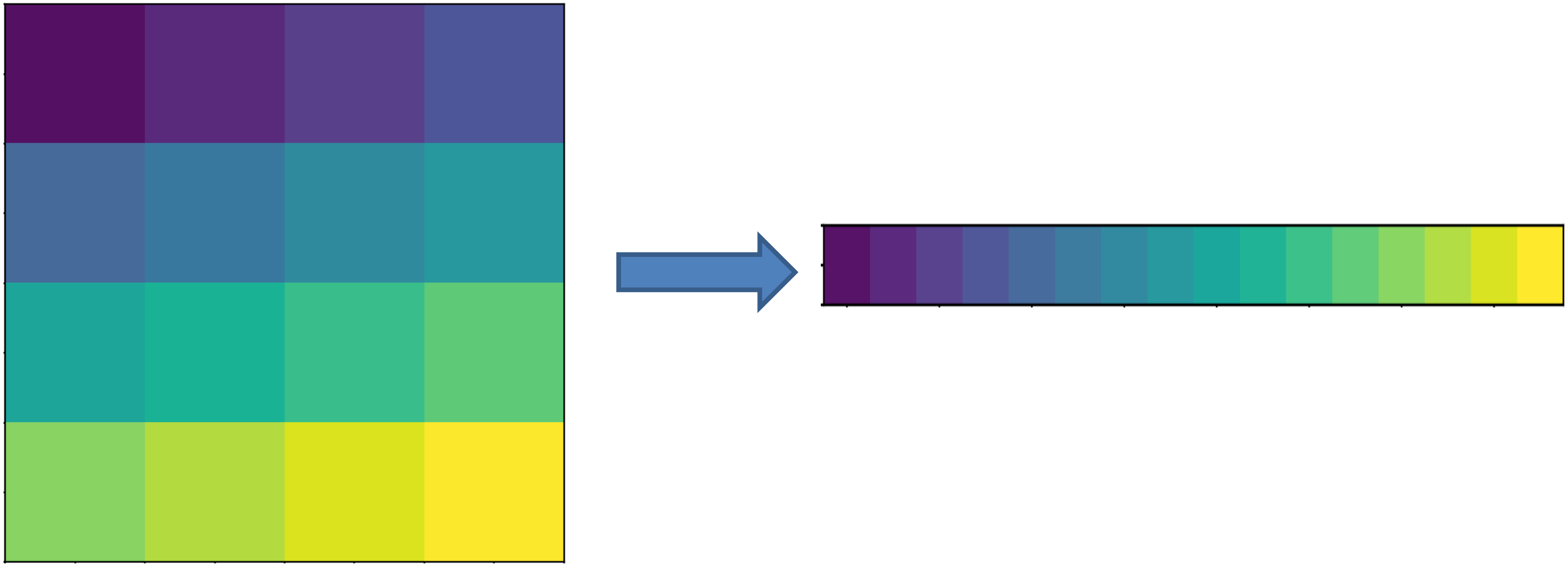
Let’s make the (almost) simplest possible ϕ

- Represent an image as a vector in $\mathbb{R}^d$
- Step 1: convert image to gray-scale and resize to fixed size



Feature space: representing images as vectors

- Step 2: Flatten 2D array into 1D vector



Let's make the simplest possible *h*

- h(x) = "dog"

Let’s make the simplest possible *h*

- h(x) = “dog”
- Okay, let’s get a little less simple than that.

Let’s make a very simple *h*

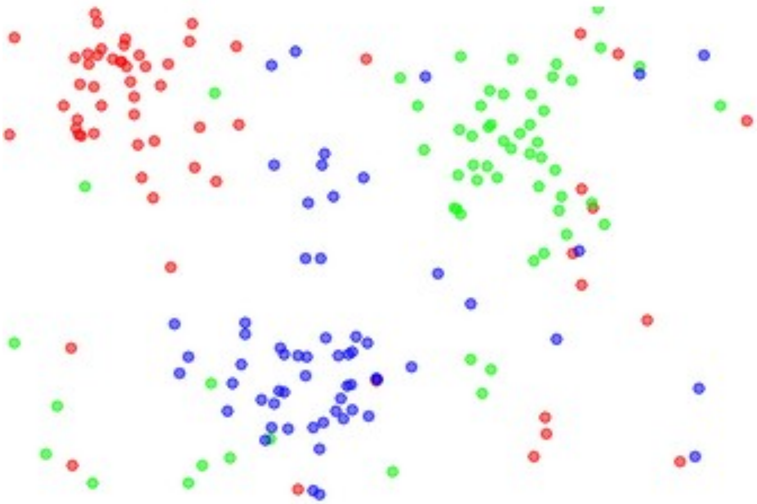
- h(x) = “dog”
- Okay, let’s get a little less simple than that.

- I’ve never seen x before, but I’ve seen a bunch of other things.

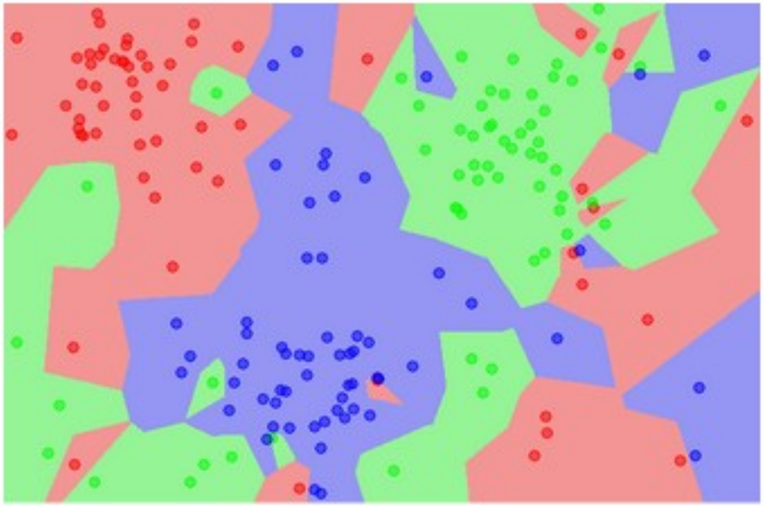
- h(x) = the label of the most similar thing to x of all the things I’ve seen.
  – assumption: **similar** data points have **similar** labels

A Simple *h*: Nearest Neighbor Classifier

the data



NN classifier



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

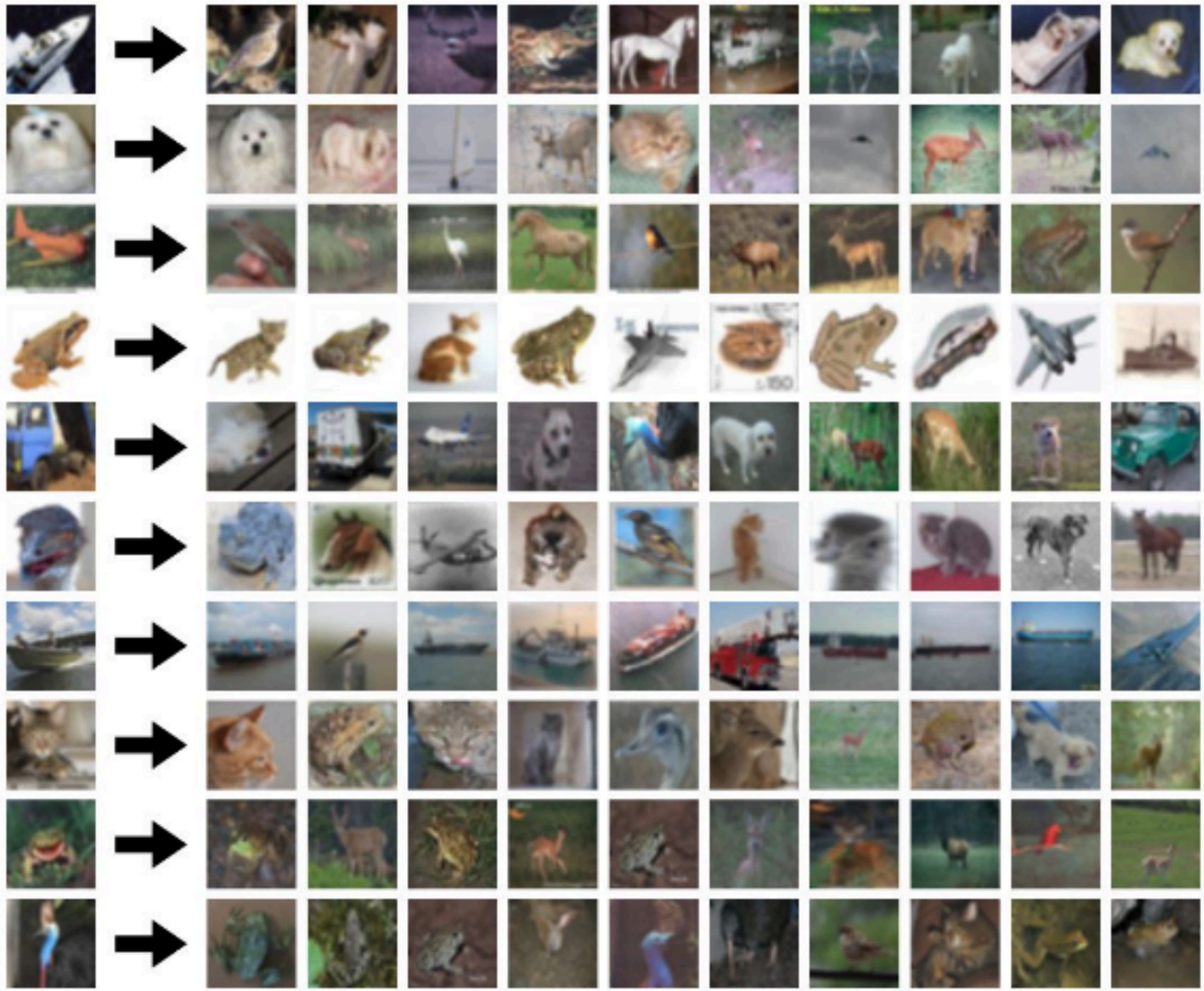
→ Memorize all data and labels

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

→ Predict the label of the most similar training image

Figures: Fei-Fei Li, Justin Johnson, & Serena Yeung

Demo:
Nearest Neighbor on MNIST



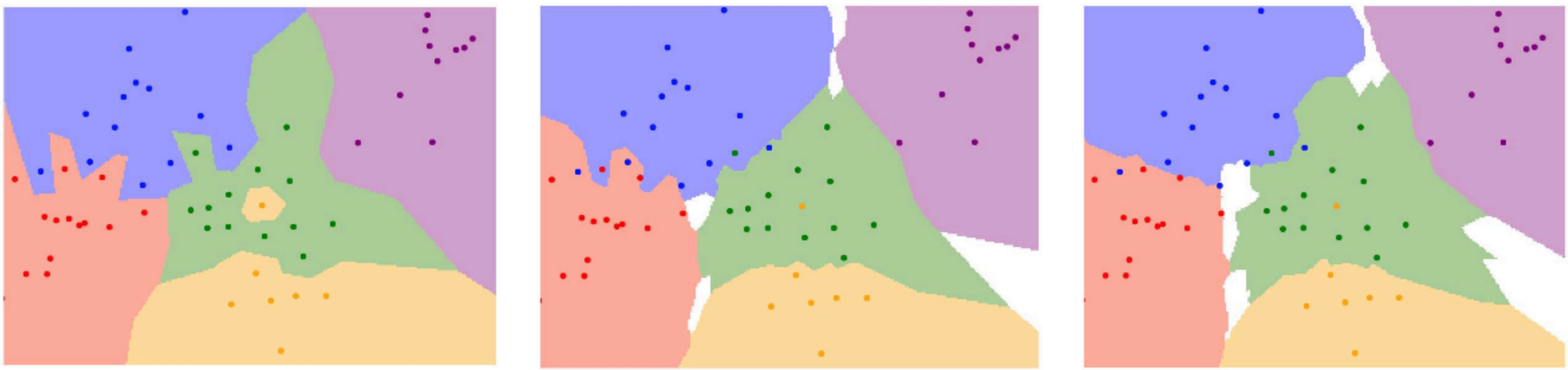
Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

An improvement: K nearest neighbors

K-Nearest Neighbors

Instead of copying label from nearest neighbor,
take **majority vote** from K closest points



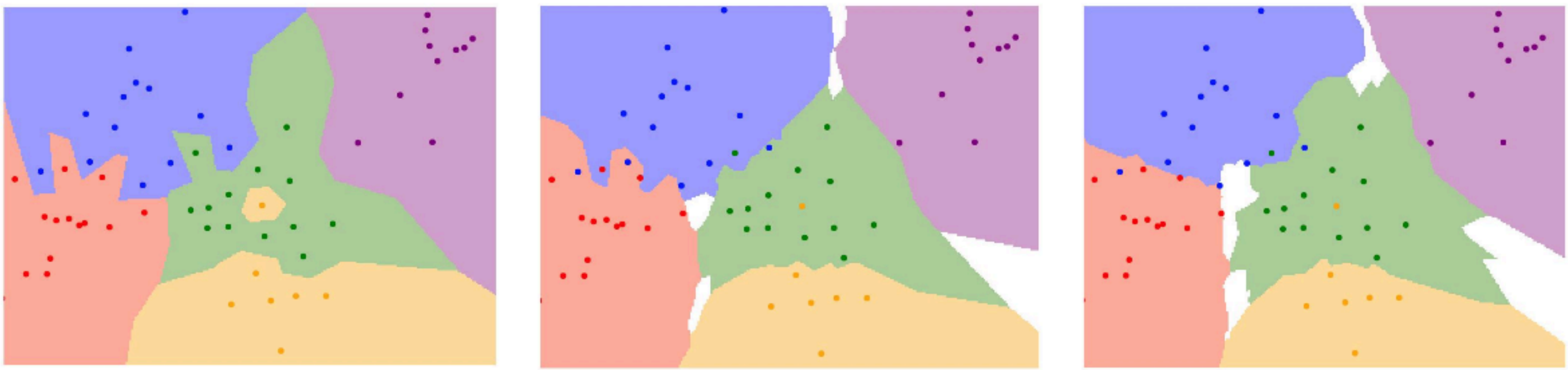
K = 1 K = 3 K = 5

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

An improvement: K nearest neighbors

K-Nearest Neighbors

Instead of copying label from nearest neighbor,
take **majority vote** from K closest points



K = 1 K = 3 K = 5

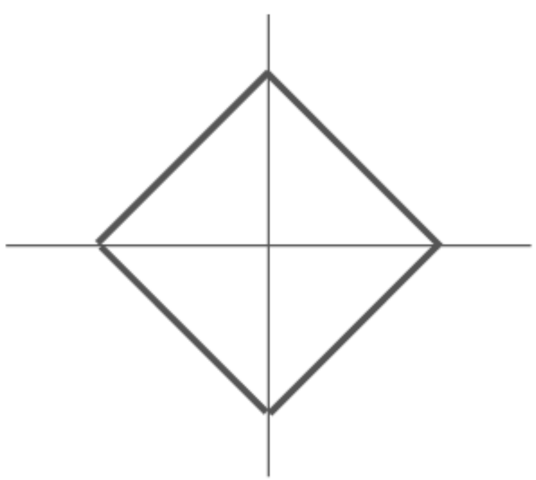
- What do we mean by "nearest" anyway?

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

K-Nearest Neighbors: Distance Metric

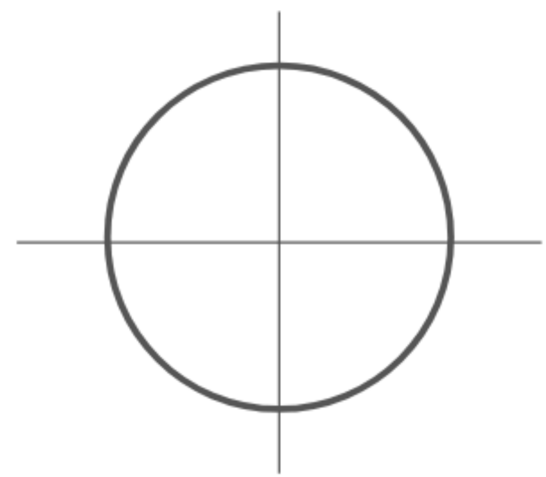
L1 (Manhattan) distance

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



L2 (Euclidean) distance

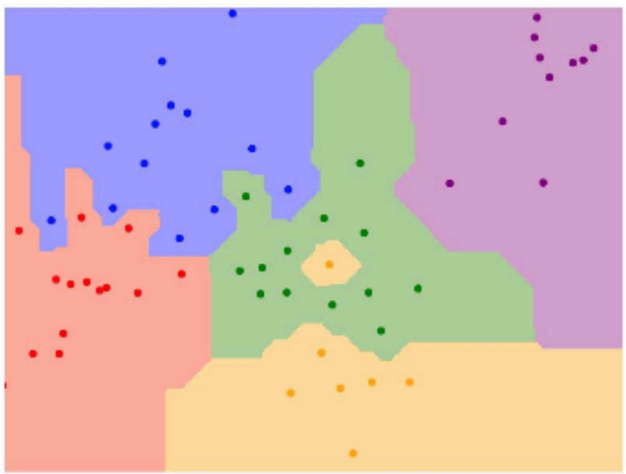
$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$



K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

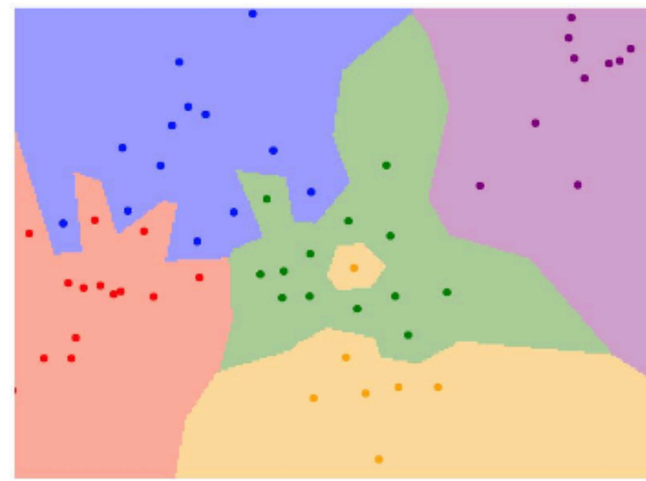
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



K = 1

L2 (Euclidean) distance

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



K = 1

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Demo

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

Simple Image Classification Algorithm

- ϕ : Convert to grayscale and unravel into a vector.

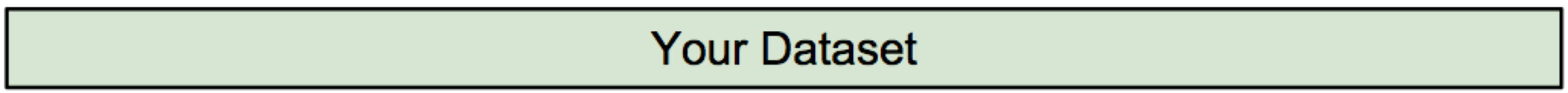
- *h*: Classify using majority label of the *k* nearest neighbors according to a distance metric *d*.

- k and d are **hyperparameters**. How do we know what to choose?
  – Depends on the problem
  – Usually no principled way to choose – trial and error is often the only way.

Setting Hyperparameters

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

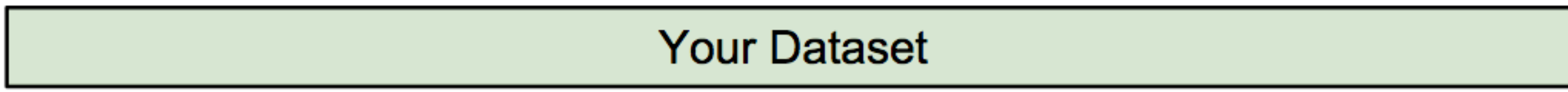
BAD: K = 1 always works perfectly on training data



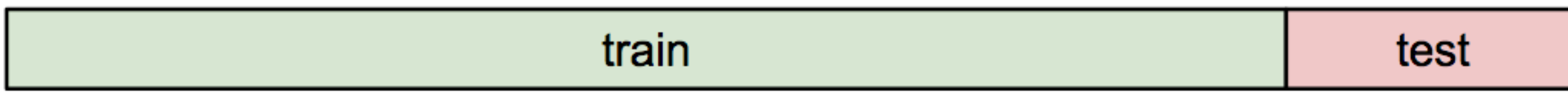
Setting Hyperparameters

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



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Setting 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

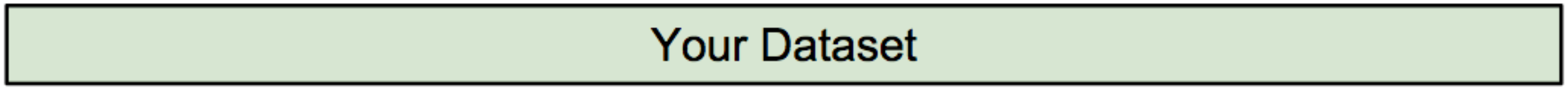
train	test

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

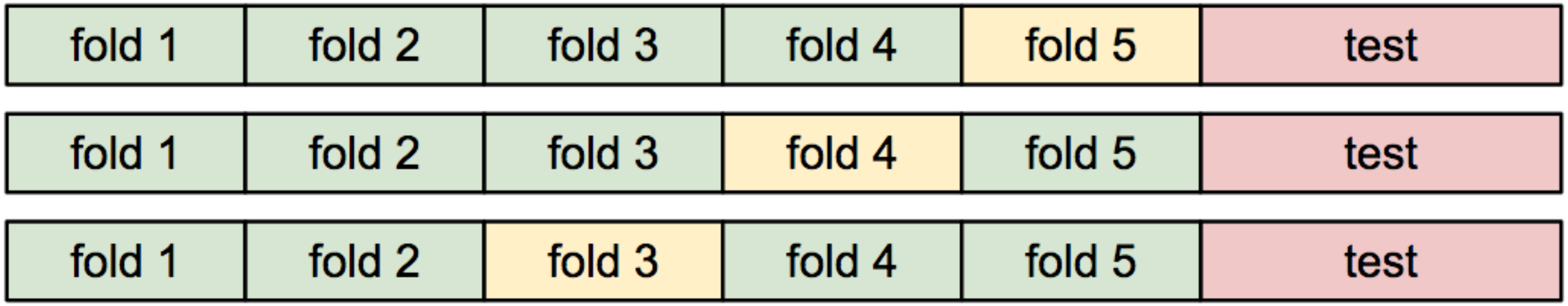
Better!

train	validation	test

Setting Hyperparameters



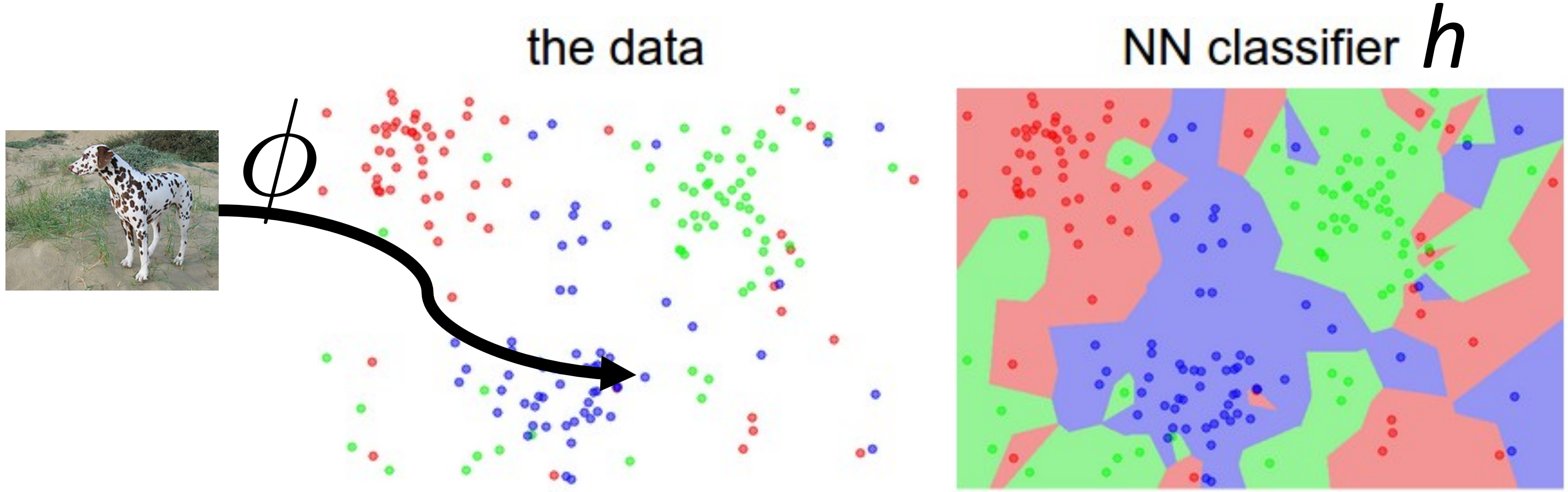
Idea #4: **Cross-Validation**: Split data into **folds**,
try each fold as validation and average the results



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

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Nearest Neighbor Classifier: Summary



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

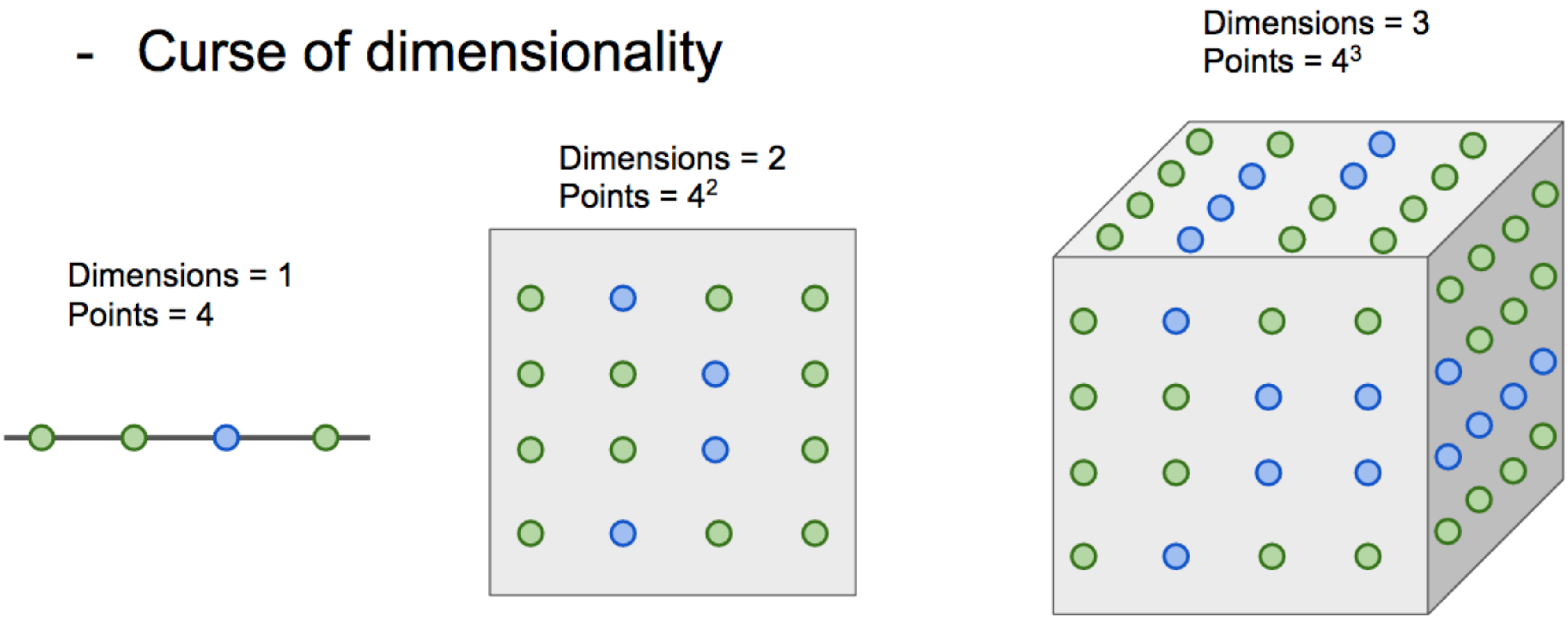
- Very slow at test time
- Distance metrics on pixels are not informative

Original | Boxed | Shifted | Tinted

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

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

- Curse of dimensionality

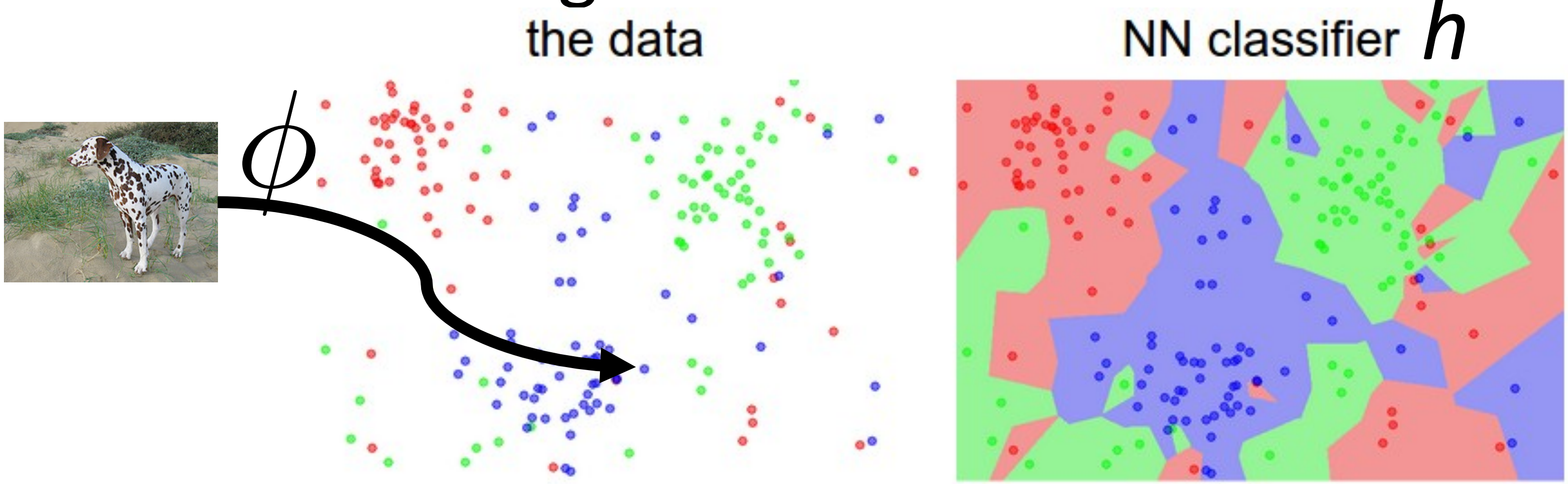


KNN: Bottom Line

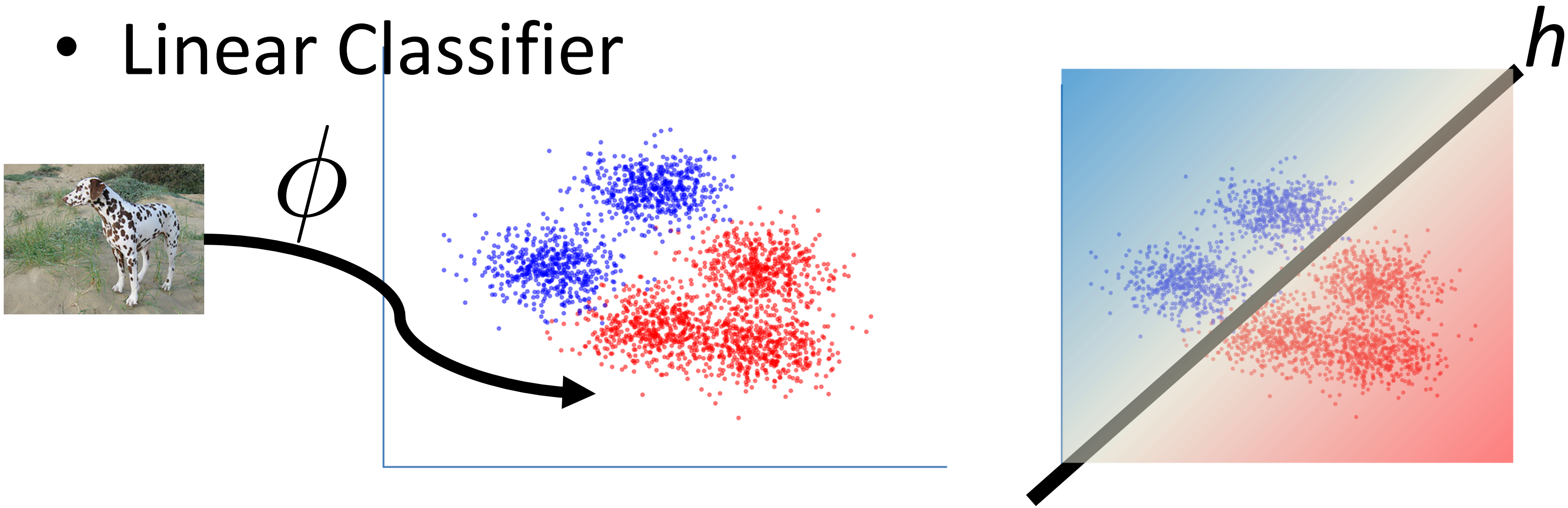
- Fast to train but slow to predict
- Distance metrics don’t behave well for high-dimensional image vectors

Classifying Images: Let’s simplify

- Nearest Neighbor Classifier



- Linear Classifier

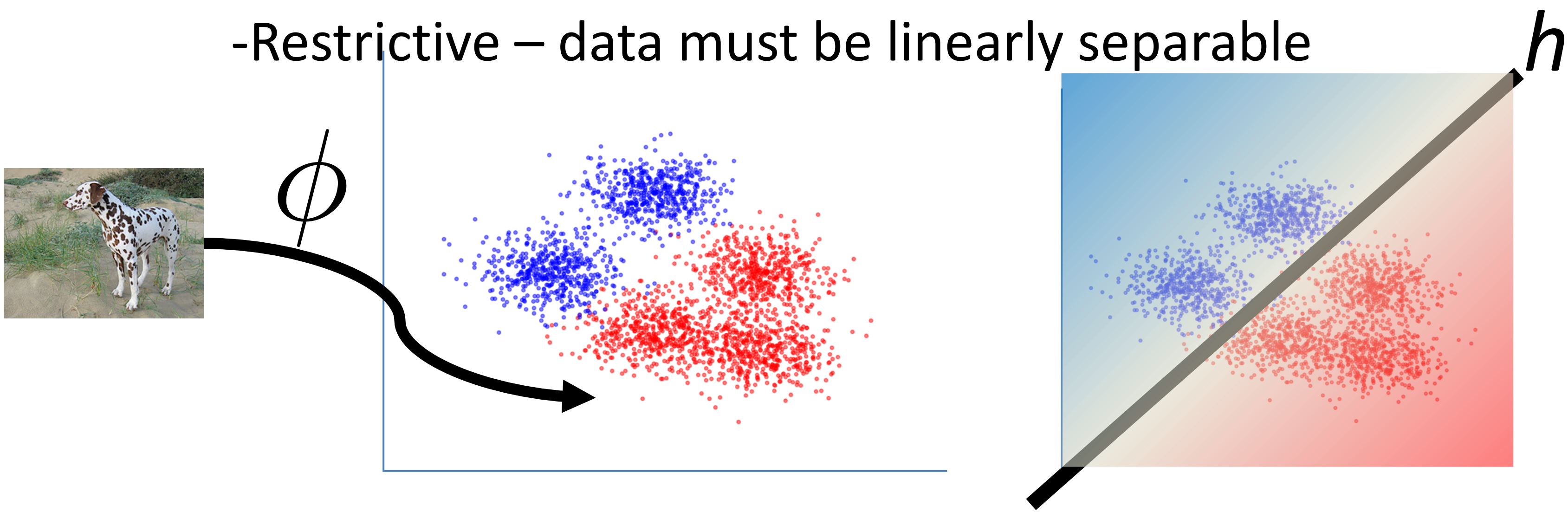


Linear classifiers

- Finding nearest neighbor is slow.
- Basic idea:
  - Training time: find a line that separates the data
  - Testing time: which side of the line is ϕ(x) on?

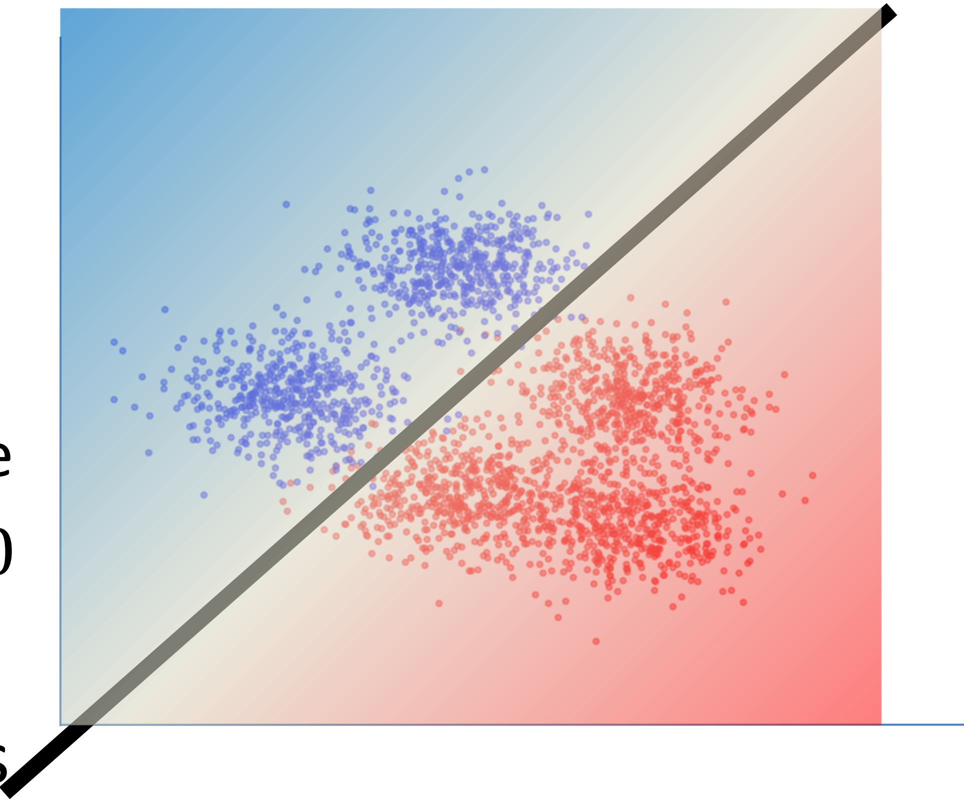
    +Fast to compute

    -Restrictive – data must be linearly separable



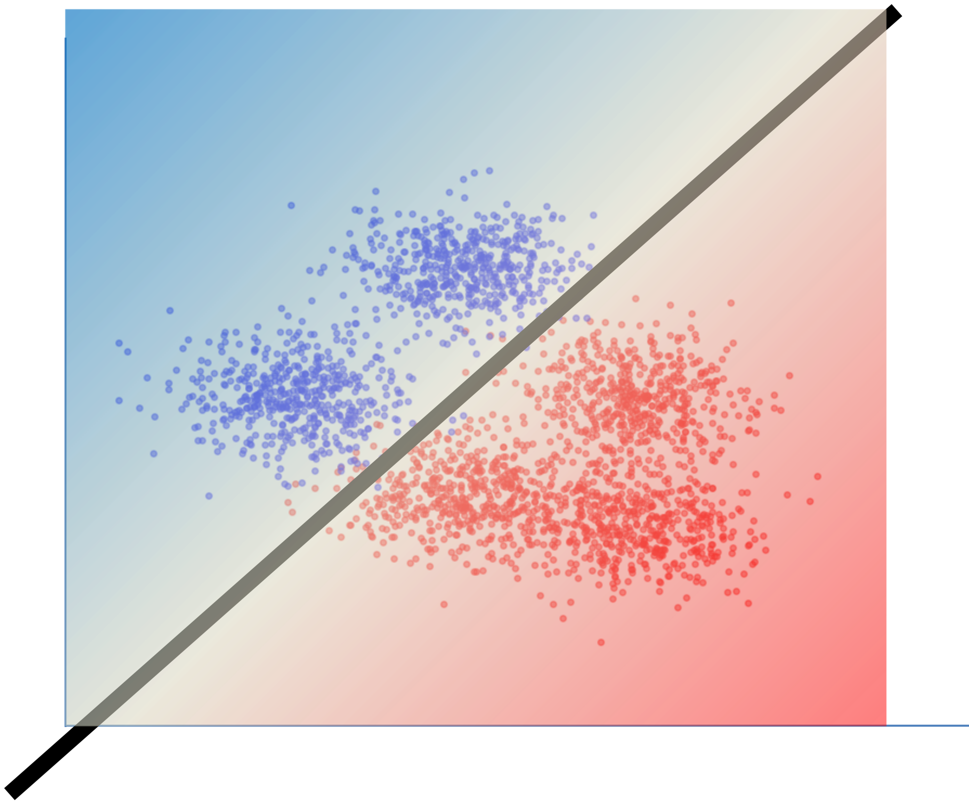
Linear classifiers

- A linear classifier corresponds to a hyperplane
  - Equivalent of a line in high-dimensional space
  - Equation: $w^T x + b = 0$
- Points on the same side are the same class



Does this ever work?

- It's easier to be linearly separable in high-dimensional space.
- But simple linear classifiers still don't work on most interesting data.



Some history from the Ante**deep**luvian Era

- Example pipeline from days of yore:
  - Detect corners and extract SIFT features
  - Collect features into a “bag of features”
  - (if you’re feeling fancy) maintain some spatial information
  - Somehow convert feature bag to fixed size
  - Apply **linear** classifier

- Key idea: ϕ is designed by hand, while *h* is learned from data.

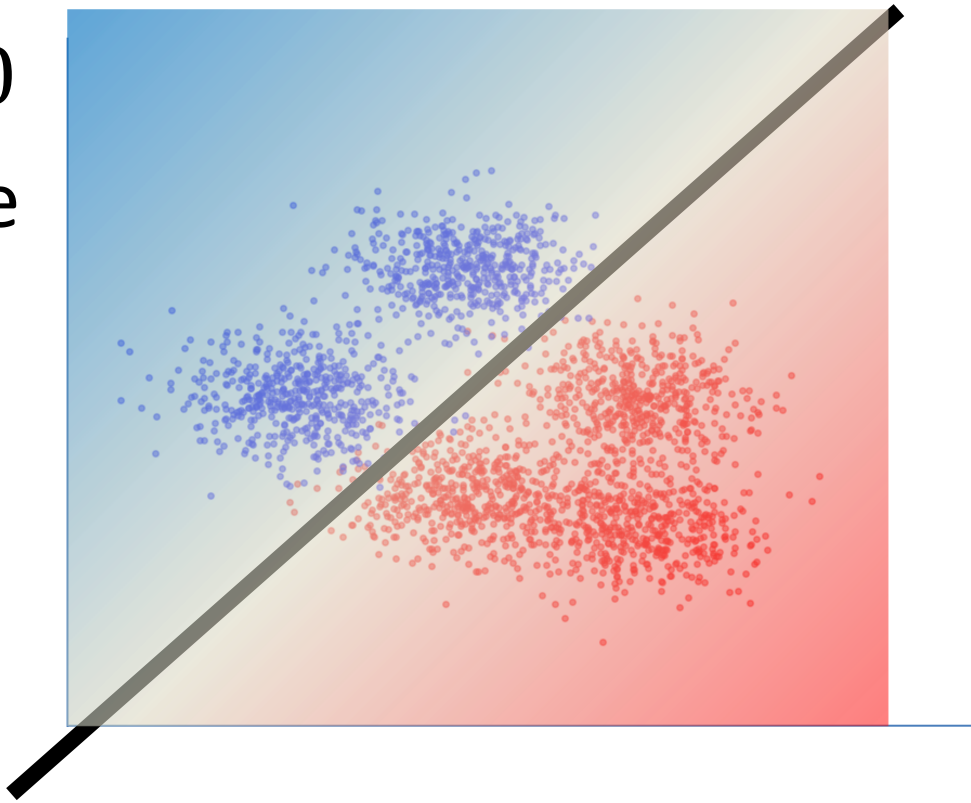
Some history of the Ante**deep**luvian Era

- Key idea: ϕ is designed by hand, while *h* is learned from data.

- Nowadays: learn both from data - “end-to-end”: image goes in, label comes out.
  - Enabled only recently by bigger
    - labeled datasets
    - compute power (GPUs)

Linear classifiers

- Equation: $w^T x + b = 0$
- Points on the same side are the same class

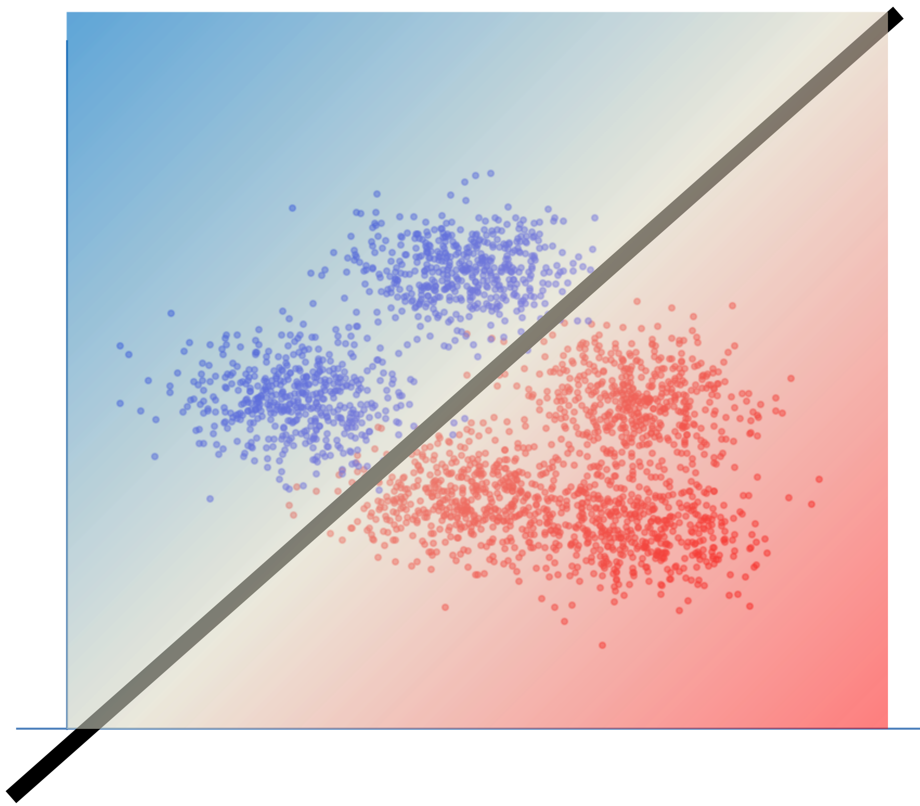


We have a classifier

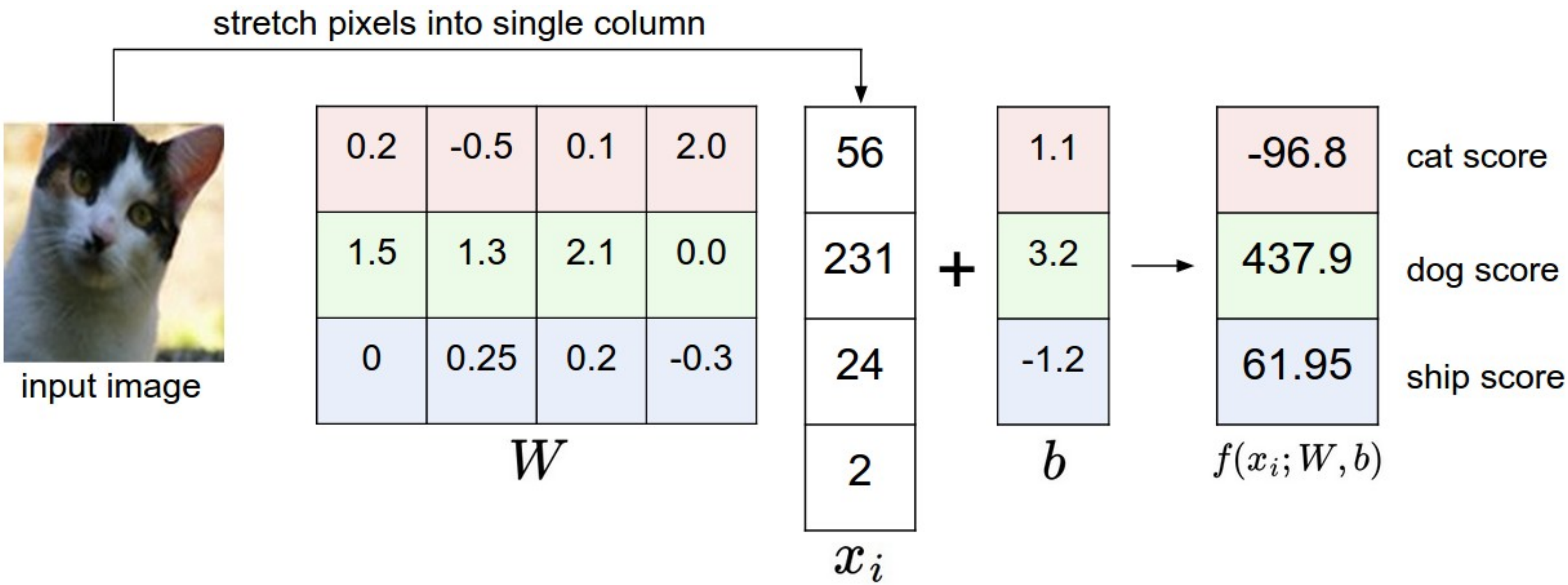
- $h(x) = w^T x + b$ gives a *score*

- Score negative: red
- Score positive: blue

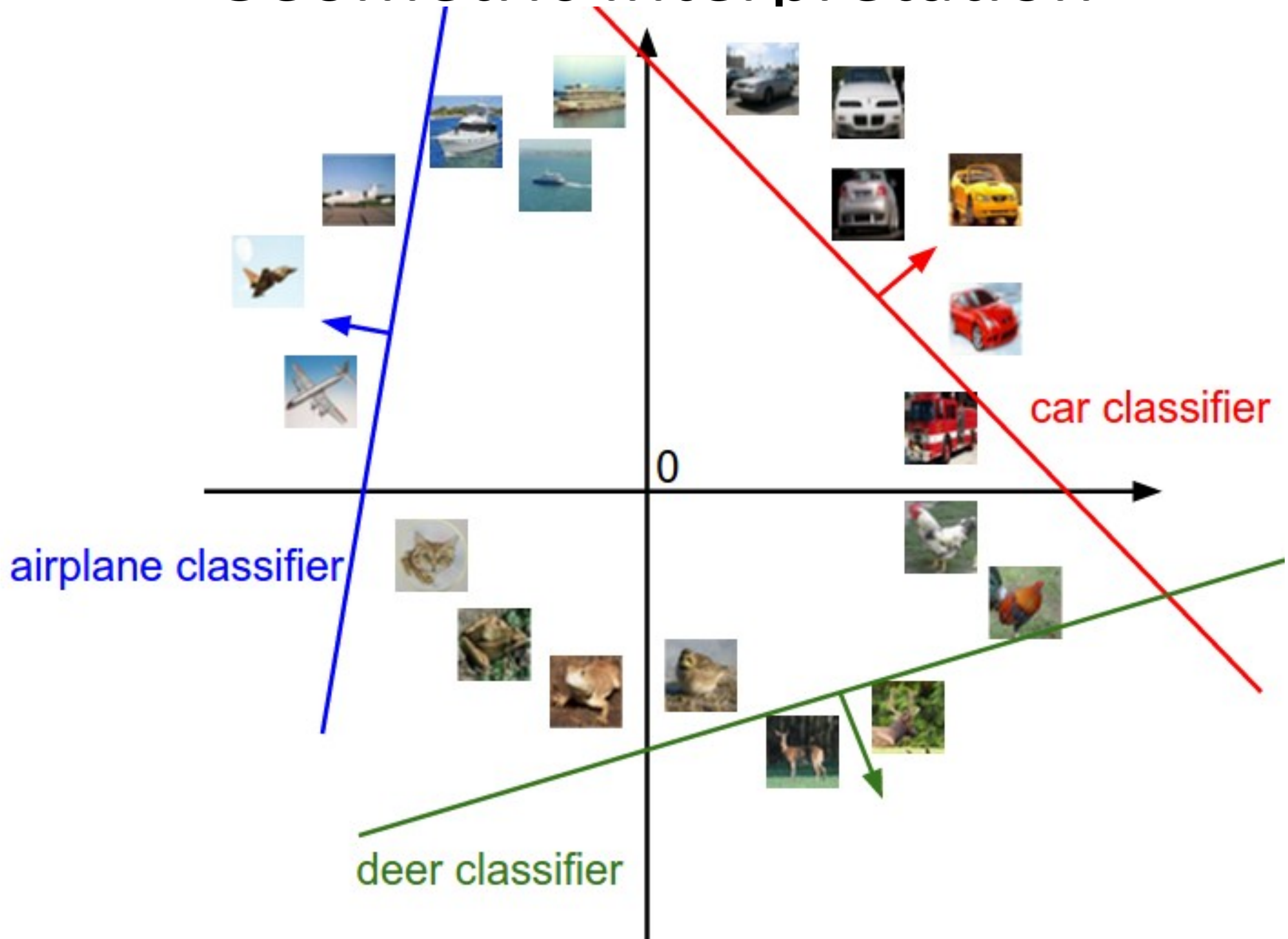
- Does it solve the runtime issues of KNN?



Multiclass Linear Classifiers: Stack multiple w^T into a matrix.



Multiclass Linear Classifier: Geometric Interpretation



The Bias Trick

The Bias Trick

- Fold b into an additional dimension of w
- Add a fixed 1 to all feature vectors.

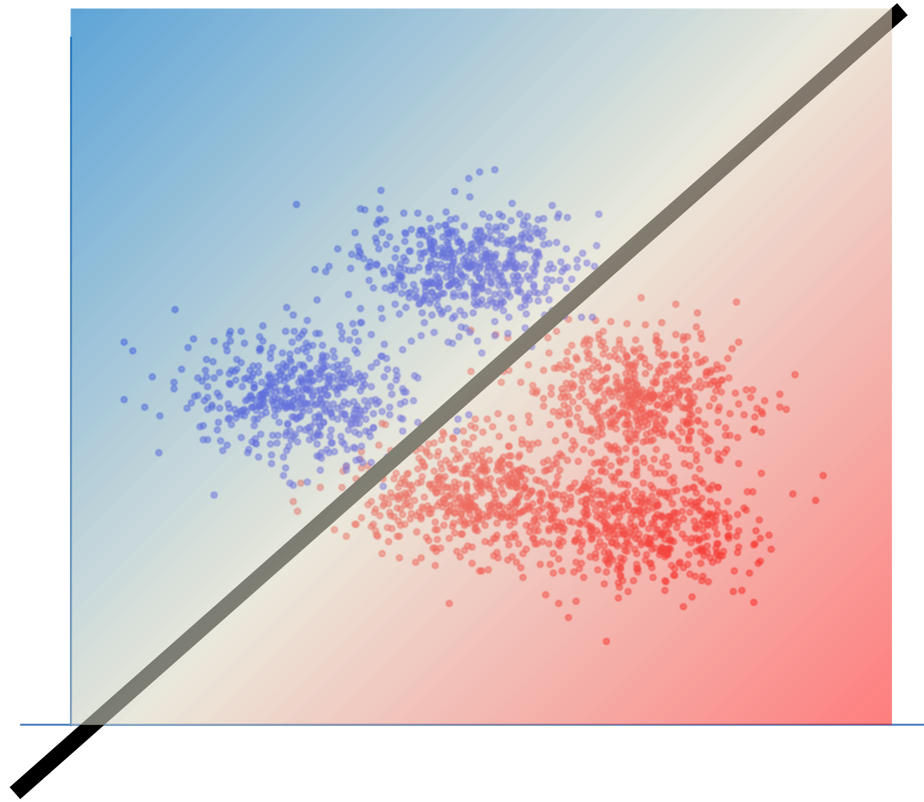
- Now, $h(x) = w^T x$

We have a classifier

- $h(x) = w^T x$ gives a *score*

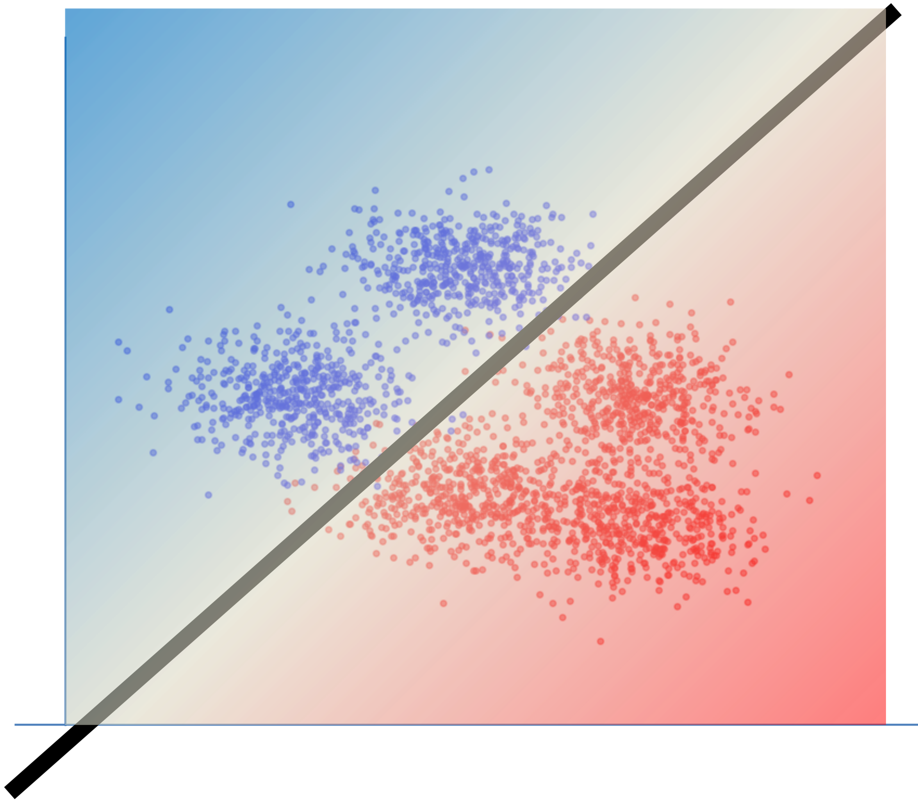
- Score negative: red
- Score positive: blue

- Where does w come from?



How do we find a good W?

- Step 1: For a given W, decide on a **Loss Function**: a measure of how much we dislike the line.
- Step 2: use **optimization** to find the W that *minimizes* the loss function.



Loss Functions

- Step 1: For a given W, decide on a **Loss Function**: a measure of how much we dislike this classifier.
- Step 2: use **optimization** to find the W that *minimizes* the loss function.
  – Linear regression: solvable in closed form
  – Useful loss functions in vision: no closed form.

Loss Functions

- Step 1: For a given W, decide on a **Loss Function**: a measure of how much we dislike this classifier.

- Loss Function intuition:
  - loss should be large if many data points are misclassified
  - loss should be small (0?) if all data is classified correctly.

Loss function: Ideas

Softmax Classifier / Cross-Entropy Loss: Intuition

$W^T x$ gives us a vector of scores, one per class (each row of W is a classifier)

Wouldn't it be nice to interpret these as probabilities?

Softmax Classifier / Cross-Entropy Loss: Intuition

$W^T x$ gives us a vector of scores, one per class (each row of W is a classifier)

Wouldn't it be nice to interpret these as probabilities?

But they're not…

- can be < 0
- don't all sum to 1

But we can treat them as **unnormalized log probabilities**.

Softmax Classifier / Cross-Entropy Loss

f = W^T x gives us a vector of scores, one per class (each row of W is a classifier)

Softmax normalization: Exponentiate to get all positive values, then normalize to sum to 1:

$$p(x_i \text{ is class } k) = \frac{e^{f_k}}{\sum_j e^{f_j}}$$

Softmax Classifier / Cross-Entropy Loss

$f = W^T x$ gives us a vector of scores, one per class (each row of W is a classifier)

Softmax normalization: Exponentiate to get all positive values, then normalize to sum to 1:

$$p(x_i \text{ is class } k) = \frac{e^{f_k}}{\sum_j e^{f_j}}$$

Cross-entropy loss: measure *KL divergence* between the **predicted** distribution and the **true** distribution:

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

Cross-Entropy Loss: Intuition

Taking stock

- We have:
  - ϕ = unravel(rgb2gray(img)), a feature extractor
  - h(x) = W^T x, a multiclass linear classifier
  - L = , a loss function

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

Taking stock

- We have:
  - ϕ = unravel(rgb2gray(img)), a feature extractor
  - h(x) = W^T x, a multiclass linear classifier
  - L = , a loss function

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

- We don't have:
  - a way to find a W that results in a small L.

Loss Functions

- Step 1: For a given W, decide on a **Loss Function**: a measure of how much we dislike this classifier.
- Step 2: use **optimization** to find the W that *minimizes* the loss function.
  - Linear regression: solvable in closed form
  - Most of the time: no closed form.

Optimization

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

How do we find a W that minimizes L?

- Bad idea: Random search.

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

How'd that go for you?

Lets see how well this works on the test set...

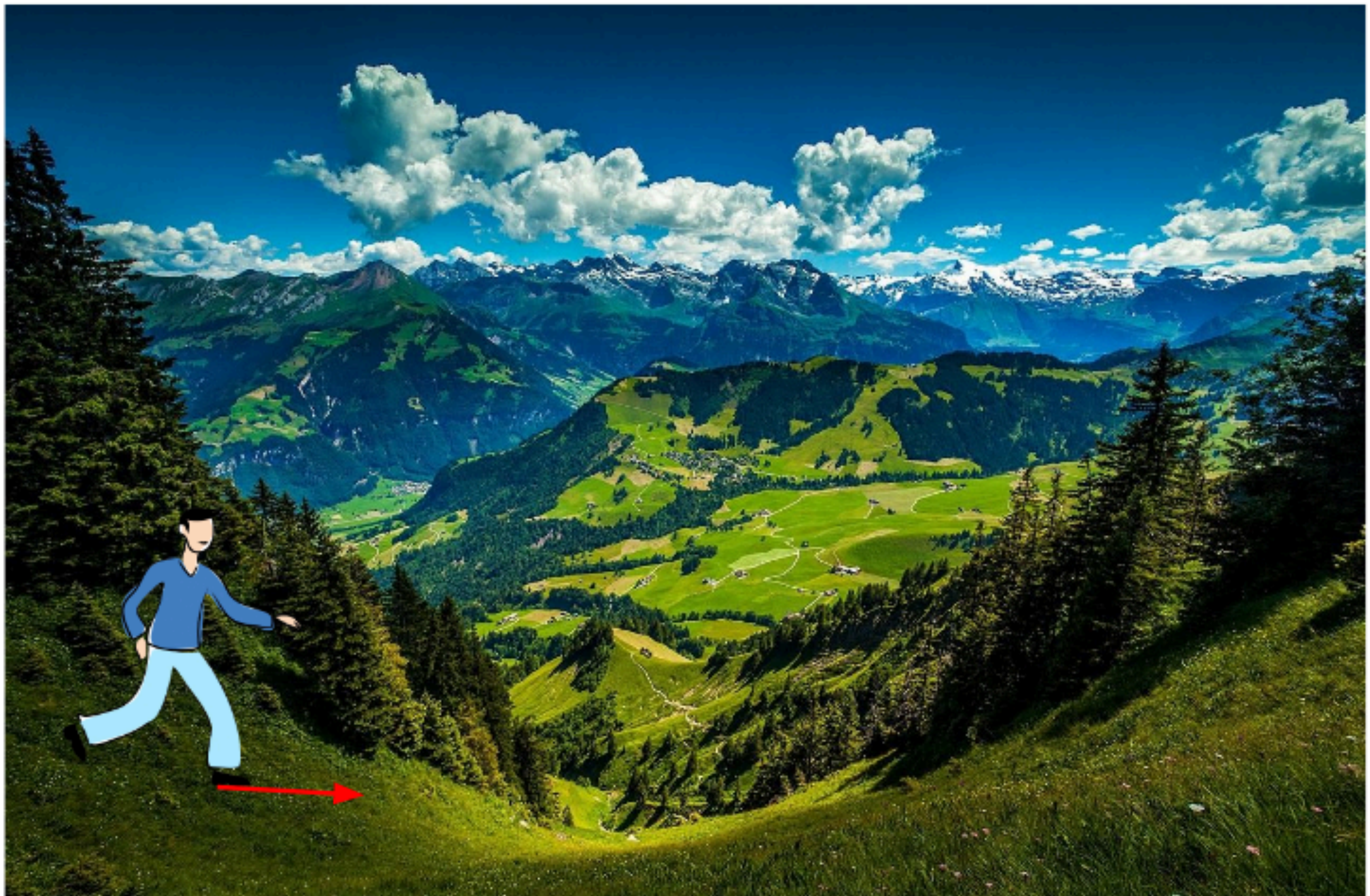
```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!
(SOTA is ~95%)

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Finding a W that minimizes L

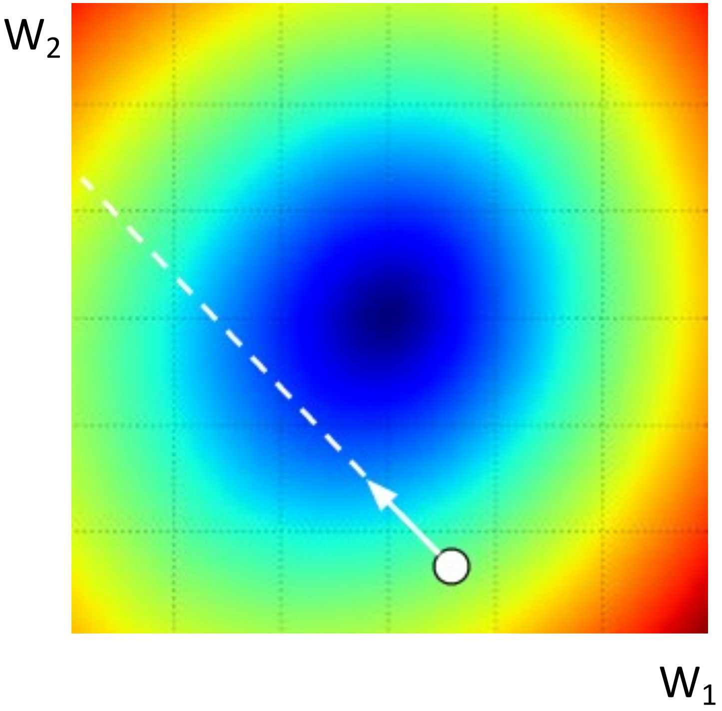
- A better idea: walk downhill.



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Gradient Descent: Generally

- Gradient of the loss function with respect to the *weights* tells us how to change the weights to improve the loss.



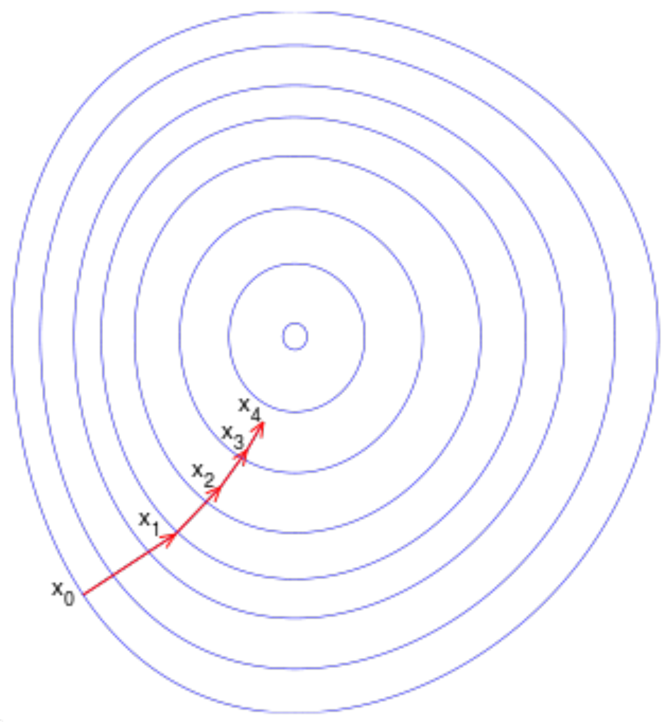
Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Gradient Descent: Intuition

Gradient Descent: Intuition

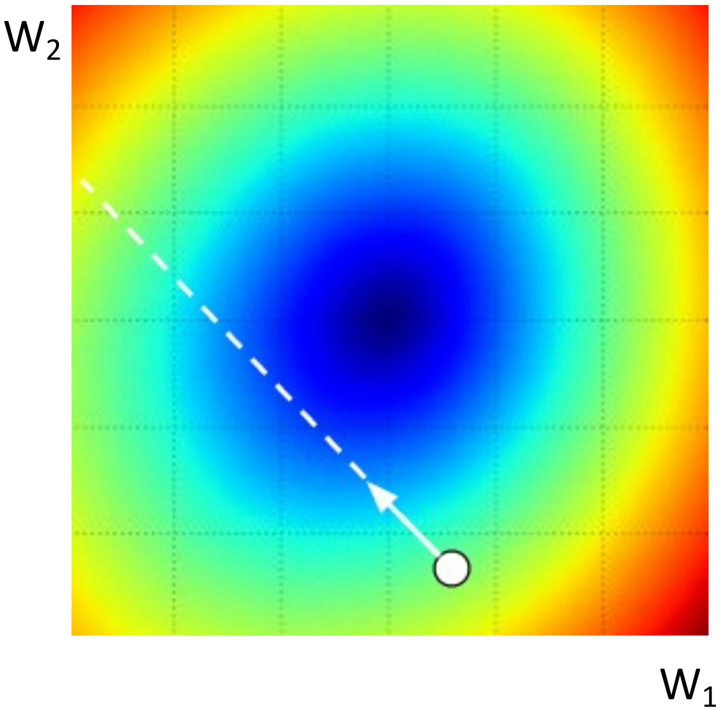


Gradient Descent: Demo

- http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/
  - select “Softmax” radio button at the bottom

Gradient Descent: Generally

- Gradient of the loss function with respect to the *weights* tells us how to change the weights to improve the loss.



- L(X; W) depends on
  - All data points $x_1..x_n$
  - Very expensive to evaluate

Stochastic Gradient Descent

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

- L(X; W) depends on
  - All data points $x_1..x_n$
  - Weights W
- Very expensive to evaluate if you have a lot of data.

Stochastic Gradient Descent

- Idea: consider only a few data points at a time.
- Loss is now computed using only a small batch (minibatch) of data points.
- Update weights the same way using the gradient of L wrt the weights.

Stochastic Gradient Descent: Intuition