CENG 506 Deep Learning

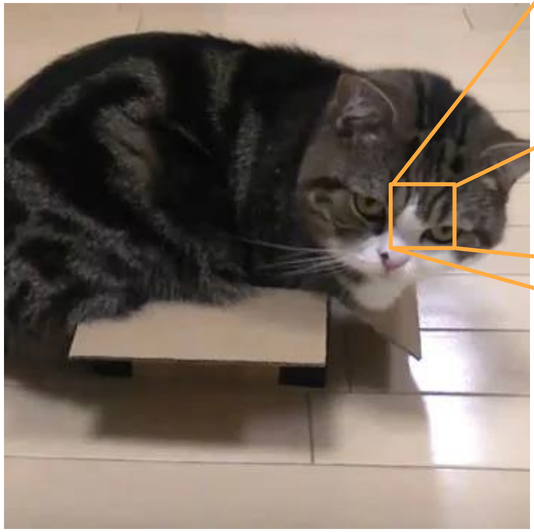
Lecture 2 - Image Classification

Image Classification

- A core task in computer vision
- Given a set of discrete labels, such as dog, cat, truck, plane, etc., our task is to predict the class of a given image.

Image Classification

- **The problem:** Semantic gap



226	241	192	99	204	44	151	221	52	37	125	118	44	26	84	8	211	10	237	248
250	47	203	53	98	195	146	159	226	115	226	227	251	51	197	104	136	158	76	16
231	201	10	177	158	130	198	27	114	239	241	203	163	76	217	191	203	33	244	2
232	48	0	66	222	73	185	52	242	242	156	9	41	251	64	97	250	202	180	1
169	212	50	123	20	214	224	254	84	178	57	115	29	2	206	205	222	24	115	168
84	88	238	233	100	183	59	118	216	118	73	127	50	217	174	163	166	127	105	102
165	238	7	249	188	15	242	151	76	253	173	242	6	8	255	0	102	165	22	38
103	219	236	59	177	93	208	8	246	71	140	14	74	237	118	64	18	195	192	232
168	186	144	170	235	134	16	206	35	161	218	3	157	46	2	104	86	134	123	111
195	74	201	48	118	71	211	200	248	241	16	96	173	113	215	173	49	205	127	48
24	218	35	50	109	179	234	18	27	143	173	165	110	93	147	101	122	239	189	82
246	132	154	255	90	96	121	50	246	250	194	219	134	48	213	186	59	202	241	175
193	202	66	69	32	153	64	89	156	174	163	125	179	197	238	57	177	33	106	200
30	198	86	161	55	157	27	140	198	6	120	215	12	207	89	221	185	12	195	153
131	31	7	64	159	82	154	101	217	35	19	40	208	144	136	151	21	37	201	112
159	77	244	124	110	74	218	73	70	145	30	160	82	122	136	159	9	155	96	239
248	250	13	16	68	236	3	125	146	70	91	114	32	188	78	171	42	4	235	13
130	209	54	100	73	197	94	135	253	110	43	98	226	154	205	16	131	237	237	198
172	63	16	18	228	56	163	222	143	149	128	196	21	127	249	79	39	156	84	177
209	141	242	6	204	36	174	202	64	223	18	16	210	213	241	252	57	170	161	107

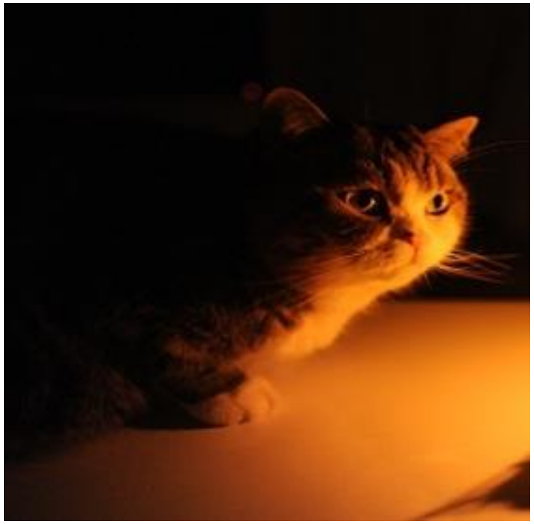
Images are represented as 3D arrays of numbers, with integers between [0,255].
Example: 300 x 100 x 3 (3 for 3 color channels in RGB)

Challenges

- Viewpoint

Challenges

- Illumination



Challenges

- Deformations

Challenges

- Occlusion

Challenges

- Background clutter

Challenges

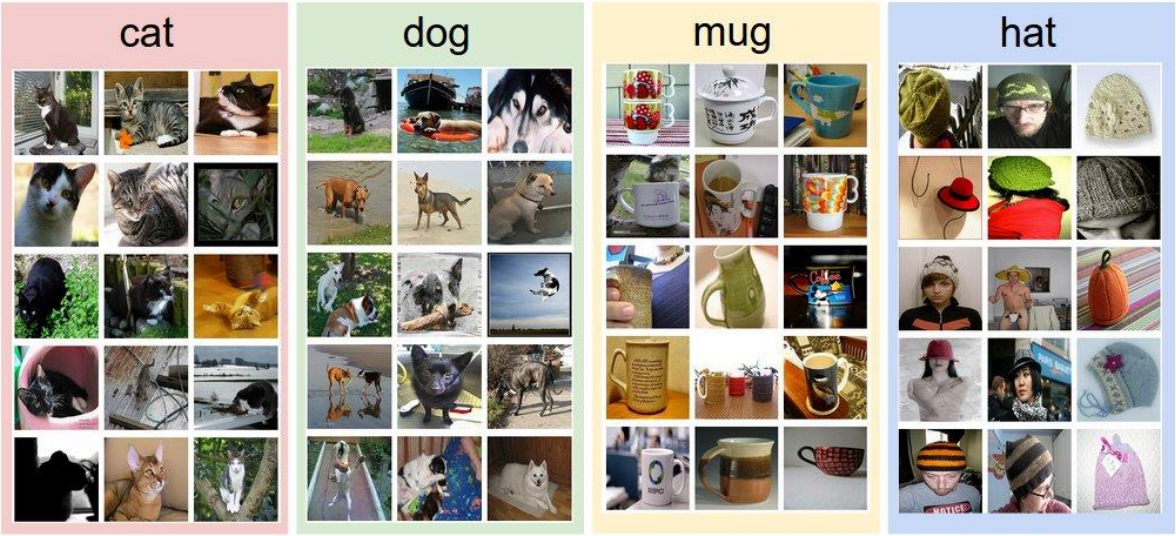
- Intra-class variation

Image Classification

- Unlike tasks like sorting or searching, there is no obvious way to hard-code the algorithm for recognizing a cat, or other classes.
- **Data-driven approach:**
    - Collect a dataset of images and labels.
    - Use Machine Learning to train an image classifier.
    - Evaluate the classifier on a withheld set of test images.

```
def train(train_imgs, train_labels):
        # build a model for images
        …
        …
        return model

def predict(model, test_imgs):
        # predict test labels
        …
        …
        return model
```



Our first approach for image classification: Nearest Neighbor Classifier

```
def train(train_imgs, train_labels):
        # build a model for images
        …
        …
        return model

def predict(model, test_imgs):
        # predict test labels
        …
        …
        return model
```

Remember all training images and labels

Predict the label of the most similar training image

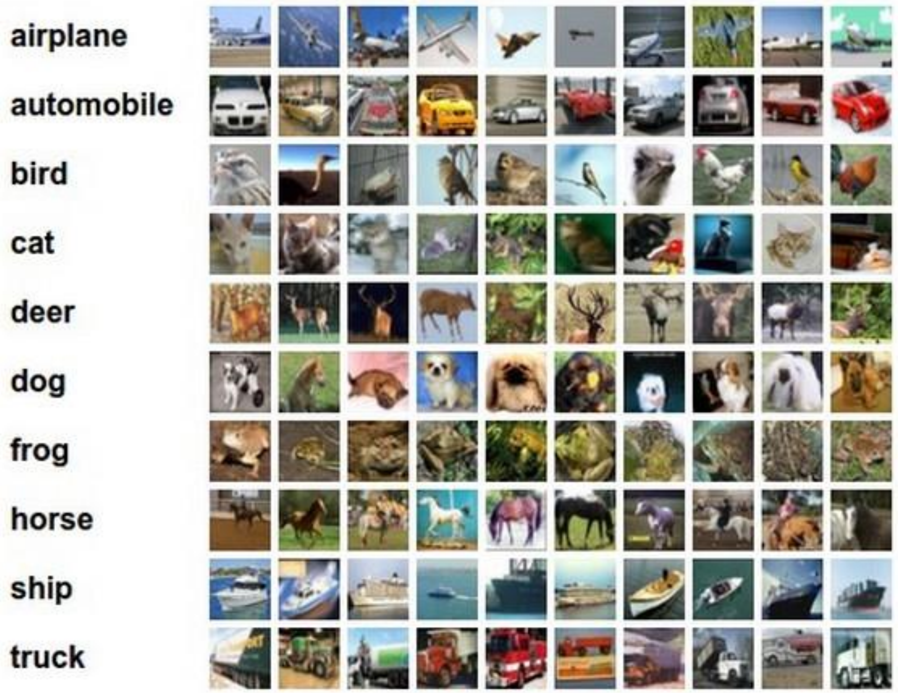
Note: This classifier has nothing to do with CNNs and rarely used in practice.

Nearest Neighbor Classifier

Example dataset: CIFAR-10

- 10 labels
- 50000 training images of size 32x32
- 10000 test images

NN Classifier Result:
For every test image (first column),
top ten nearest neighbors in rows:



Nearest Neighbor Classifier

- The notion of being the “nearest” brings about a comparison among the images.
- So, how do we compare the images? What is the **distance metric**?

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

$$\left| \overset{\text{test image}}{\begin{array}{|c|c|c|c|} \hline 56 & 32 & 10 & 18 \\ \hline 90 & 23 & 128 & 133 \\ \hline 24 & 26 & 178 & 200 \\ \hline 2 & 0 & 255 & 220 \\ \hline \end{array}} - \overset{\text{training image}}{\begin{array}{|c|c|c|c|} \hline 10 & 20 & 24 & 17 \\ \hline 8 & 10 & 89 & 100 \\ \hline 12 & 16 & 178 & 170 \\ \hline 4 & 32 & 233 & 112 \\ \hline \end{array}} \right| = \overset{\text{pixel-wise absolute value differences}}{\begin{array}{|c|c|c|c|} \hline 46 & 12 & 14 & 1 \\ \hline 82 & 13 & 39 & 33 \\ \hline 12 & 10 & 0 & 30 \\ \hline 2 & 32 & 22 & 108 \\ \hline \end{array}} \rightarrow 456$$

Nearest Neighbor Classifier

For every test image:

- Find nearest train image with L1 distance
  - calculate the distances from the test image to all training images
  - find the minimum of these distances
- Predicted label is the label of nearest training image

Python code for reference:

```
Xtr, Ytr, Xte, Yte = load_CIFAR10('data/cifar10/') # a magic function we provide
# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes 50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes 10000 x 3072
```

```
nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test images
# and now print the classification accuracy, which is the average number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )
```

Nearest Neighbor Classifier

NearestNeighbor class for reference

```
class NearestNeighbor(object):
  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

- The classification speed depends **linearly** on the size of training data!
- This is the opposite of what we like:
    - test time performance is usually much more important in practice.
    - time spent for training is less critical.
    - CNNs flip this: expensive training, cheap test evaluation
- The choice of distance is a **hyperparameter**. Common choices are:

L1 (Manhattan) distance

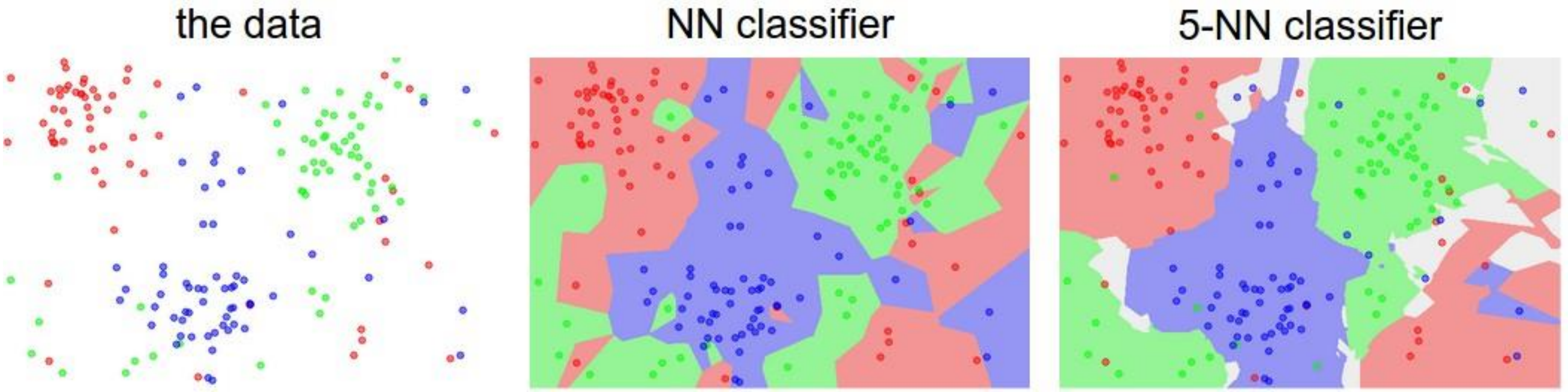
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 (Euclidean) distance

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

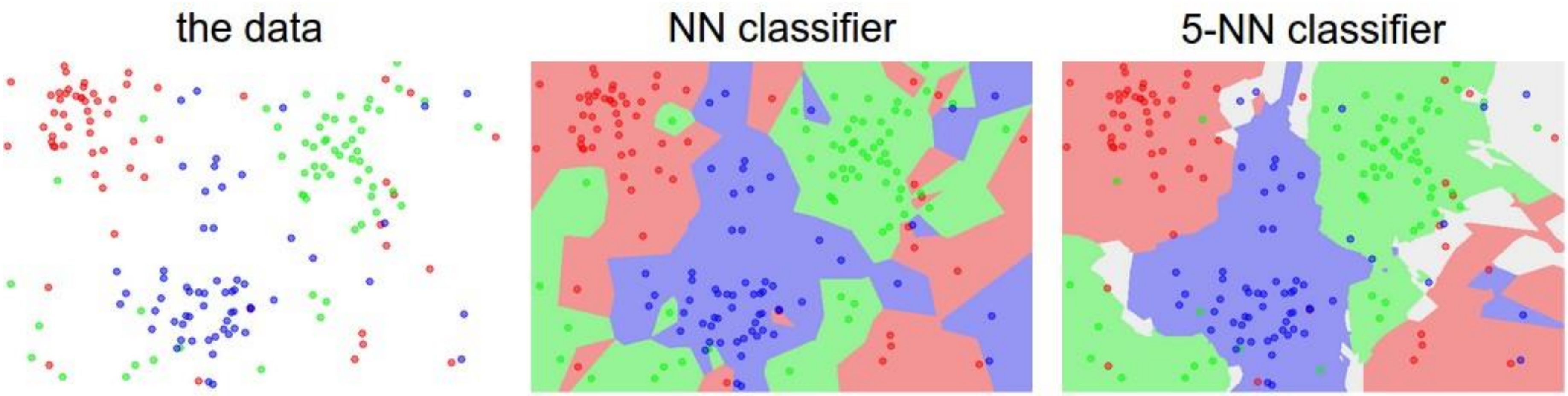
k-Nearest Neighbor Classifier

- Find the k nearest images
  - k is a user-defined constant
- Have them vote on the label
- An unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the k training samples nearest to that query point.



k-Nearest Neighbor Classifier

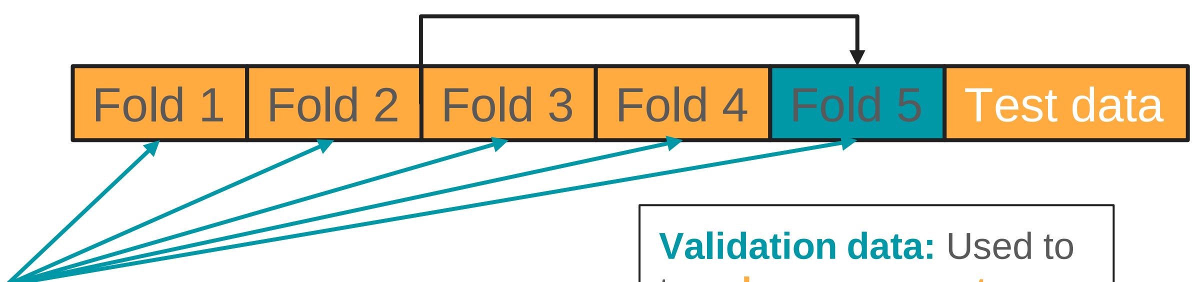
- What can you tell about the accuracy of the k - nearest neighbor classifier on the training data, when k = 1 and k = 5?



k-Nearest Neighbor Classifier

- How do we set the hyperparameters?
  - What is the best distance to use?
  - What is the best value of k to use?
- Very problem-dependent.
  - Must try them all out and see what works best.
  - Best on test set?

  - **Very bad idea!** The test set is a proxy for the generalization performance!
  - Use only **VERY SPARINGLY**, at the end!

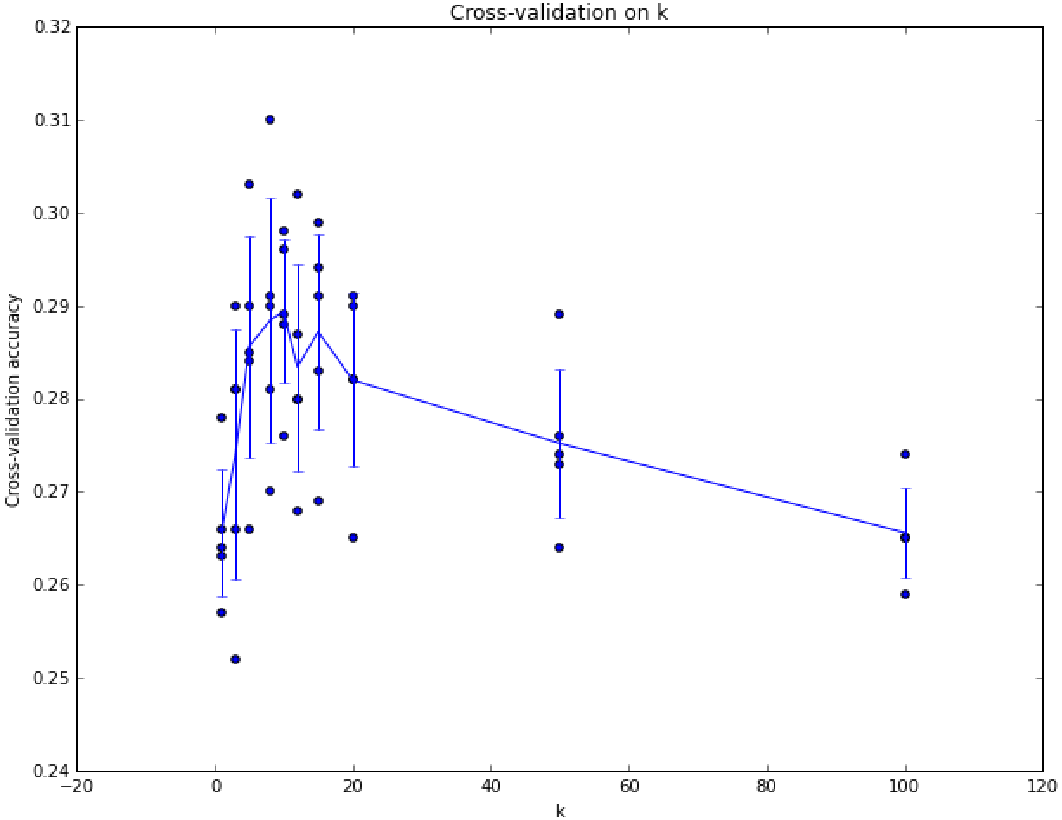


Validation data: Used to tune **hyperparameters**

Cross-validation: Cycle through the folds for validation and average the results.

k-Nearest Neighbor Classifier

- Example of 5-fold cross-validation for the value of k
- Each point: single outcome
- The line goes through the mean, bars indicate standard deviation

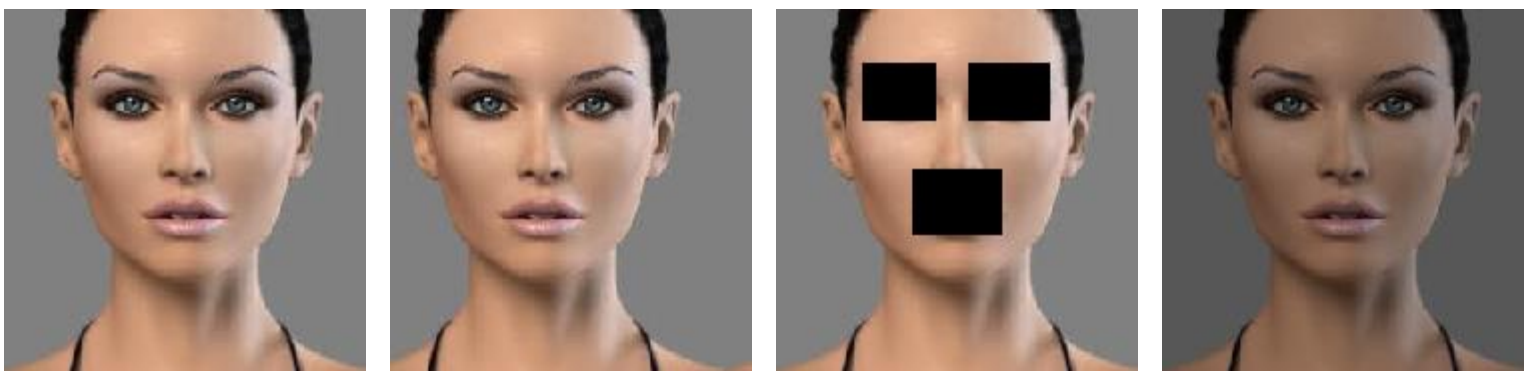


Seems that k ~= 7 works best for this data

k-Nearest Neighbor Classifier

- Not very useful with images!
  - Terrible performance at test time
  - Distance metrics on level of whole images can be very unintuitive

original | shifted | messed up | darkened

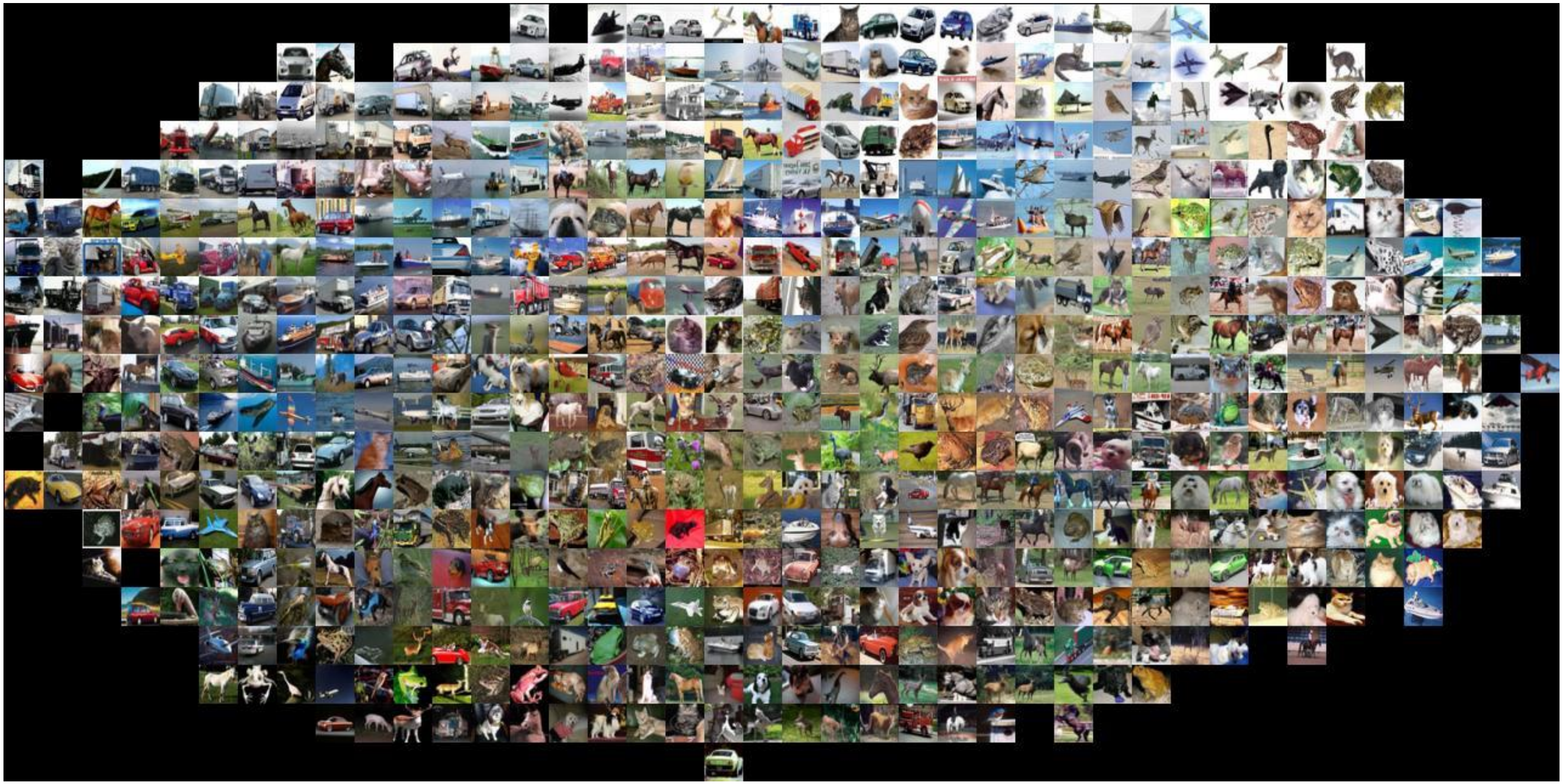


All 3 images have the same L2 distance to the original one.

Summary

- **Image Classification:** Given a training set of labeled images, we are asked to predict labels on a test set. Accuracy of the predictions (fraction of correctly predicted images) is reported.
- The **k-Nearest Neighbor Classifier** is introduced, which predicts the labels based on nearest images in the training set
- The choice of distance and the value of k are hyperparameters that are tuned using a validation set, or through cross-validation if the size of the data is small.
- Once the best set of hyperparameters is chosen, the classifier is evaluated once on the test set, and reported as the performance of kNN on that data.

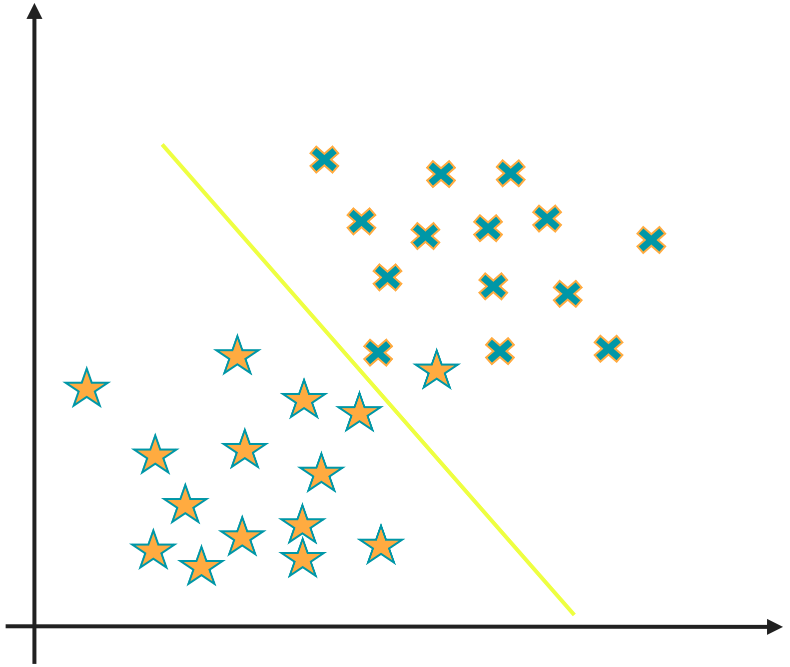
Summary



Images that are nearby in this image are considered to be near based on the L2 distance.
Notice the strong effect of background rather than semantic class differences.

Our second approach for image classification: Linear Classification

- In the graph below, every point is described by two features.
- In building mathematical models for classifying, if we focus on dividing these points with a straight line, this is called **linear classification**.



A linear classifier makes a classification decision based on the value of a **linear combination** of the characteristics, which are also known as feature values.

Linear Classification

- It is parametric approach, we are going to build a model (a classifier).
- We will no longer need to *remember* all of the training data.

image   parameters (or weights)   bias

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

10 numbers indicating class scores

Image size: [32x32x3]
A vector of length 3072

10 x 3072   3072 x 1   10 x 1

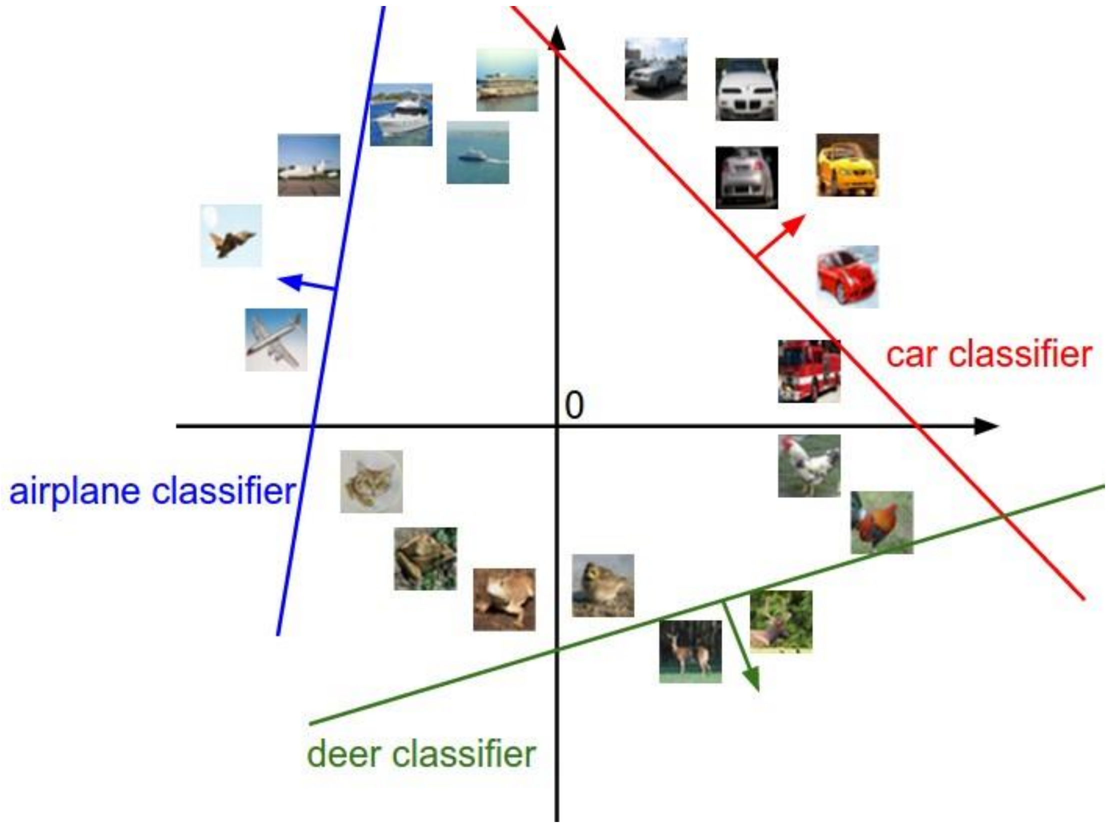
Linear Classification

- Example with a 2x2 image and 3 classes:

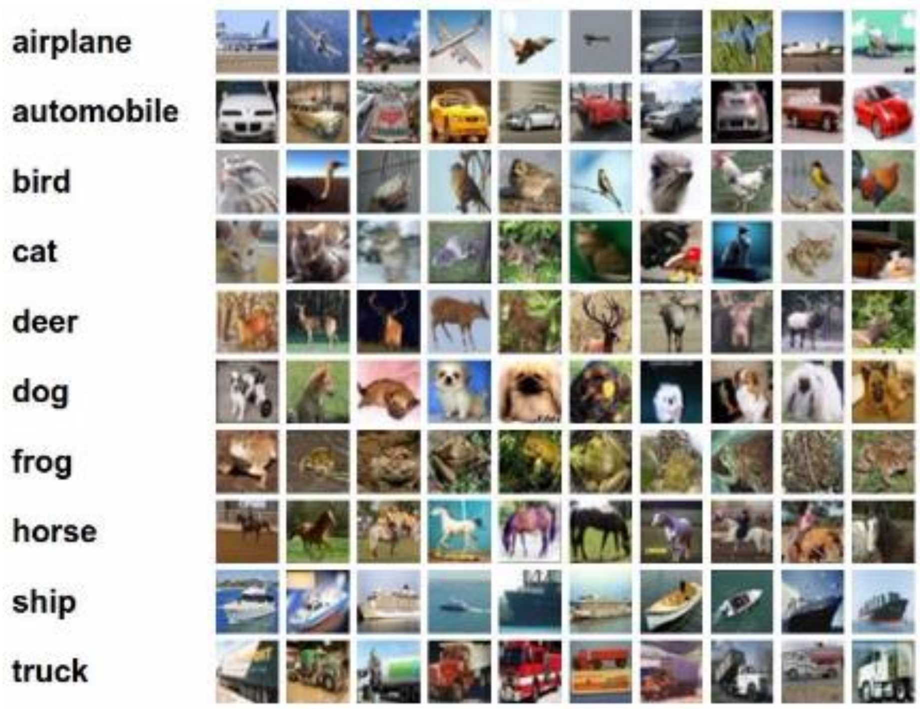
$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0.0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W} \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{\mathrm{b}} = \underbrace{\begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}}_{\mathrm{f}(x,W,\mathrm{b})} \begin{array}{l} \text{Class 1 score} \\ \text{Class 2 score} \\ \text{Class 3 score} \end{array}$$

Linear Classification

- What does a linear classifier do?
- What would be a hard set of classes for a linear classifier to distinguish?



Linear Classification Result on CIFAR-10



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

A linear score function

Example trained weights of a linear classifier trained on CIFAR-10:

Linear Classification

- How do we find the “best” parameters?
- What do we mean by the “best” parameters?

- We have to define a **loss function** that quantifies how satisfied we are with the scores across the training data.
- We have to come up with a way of efficiently finding the parameters that minimize the loss function, which is called **optimization**.

Linear Classification

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Multiclass SVM loss:

Given an example (x_i, y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i, W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Linear Classification

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
loss	2.9		

Multiclass SVM loss:

Given an example (x_i, y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i, W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max (0, 5.1-3.2+1) +max(0, -1.7-3.2+1)
= max(0, 2.9) + max(0, -3.9) = 2.9 + 0 = 2.9

Linear Classification

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



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
loss	2.9	0	

Multiclass SVM loss:

Given an example (x_i, y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i, W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max (0, 1.3-4.9+1) +max(0, 2.0-4.9+1)
= max(0, -2.6) + max(0, -1.9) = 0 + 0 = 0

Linear Classification

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



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
loss	2.9	0	10.9

Multiclass SVM loss:

Given an example (x_i,y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i,W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max (0, 2.2-(-3.1)+1) +max(0, 2.3-(-3.1)+1)
= max(0, 5.3) + max(0, 5.6) = 5.3 + 5.6 = 10.9

Linear Classification

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
loss	2.9	0	10.9

Multiclass SVM loss:

Given an example (x_i, y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i, W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

and the full training loss is the mean over all examples in the training data:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

= (2.9 + 0 + 10.9) / 3

= **4.6**

Linear Classification

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

- What if the sum was instead over all classes? (including $j = y_i$)
- What if we used a mean instead of a sum?
- What if we used the square of the max value?
- What is the min/max possible loss?
- Usually at initialization W are small numbers, so all s ~= 0. What is the loss?

Linear Classification

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Example numpy code:

```
def L_i_vectorized(x, y, W):
  scores = W.dot(x)
  margins = np.maximum(0, scores - scores[y] + 1)
  margins[y] = 0
  loss_i = np.sum(margins)
  return loss_i
```

Linear Classification

- $f(x, W) = Wx$

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

- There is a problem!
- Suppose that we found a W such that L = 0. Is this W unique?
- When L = 0, what happens if we multiply all elements of W by 2?

Linear Classification

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

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
loss	2.9	0	

Multiclass SVM loss:

Given an example (x_i, y_i):

- where x_i is the image
- where y_i is the (integer) label,
- and using the shorthand $s = f(x_i, W)$ for the scores vector,

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max (0, 1.3-4.9+1) +max(0, 2.0-4.9+1)
= max(0, -2.6) + max(0, -1.9) = 0 + 0 = 0

With W twice as large:

= max (0, 2.6-9.8+1) +max(0, 4.0-9.8+1)
= max(0, -6.2) + max(0, -4.8) = 0 + 0 = 0

Linear Classification - Regularization

regularization strength (hyperparameter)

- **Weight regularization:**

$$L = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1)}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

- L2 regularization $\longrightarrow$ $R(W) = \sum_k \sum_l W_{k,l}^2$
- L1 regularization $\longrightarrow$ $R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic net (L1 + L2) $\longrightarrow$ $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$
- Max norm regularization
- Dropout

Linear Classification - Regularization

- Regularization tends to improve generalization. It means that no input dimension can have a very large influence on the scores all by itself.
- Motivation (L2 regularization example)

$$\left.\begin{array}{l} x = [1,1,1,1] \\ W_1 = [1,0,0,0] \\ W_2 = [0.25,0.25,0.25,0.25] \end{array}\right\} W_1^{\top}x = W_2^{\top}x = 1$$

- L2 penalty of W_1 is 1.0 while the L2 penalty of W_2 is only 0.25.
- Since the weights in W_2 are smaller and more diffuse, the final classifier is encouraged to take into account all input dimensions to small amounts rather than a few input dimensions and very strongly.

Softmax Classifier

- Softmax classifier brings a loss alternative to SVM loss
- Scores serve as unnormalized log probabilities of the classes

$$s = f(x_i; W)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax function

- We want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$L_i = -\log P(Y = y_i | X = x_i)$$

- In summary:

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

Softmax Classifier

scores = unnormalized log probabilities of the classes

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

cat	**3.2**		**24.5**		**0.13**	$\longrightarrow$ L_i = -log(0.13) = 0.89
car	5.1	$\xrightarrow{\text{exp}}$	164.0	$\xrightarrow{\text{normalize}}$	0.87	
frog	-1.7		0.18		0.00	
	unnormalized log probabilities		unnormalized probabilities		normalized probabilities	

Realize that softmax classifier also has a probabilistic interpretation

Softmax Classifier

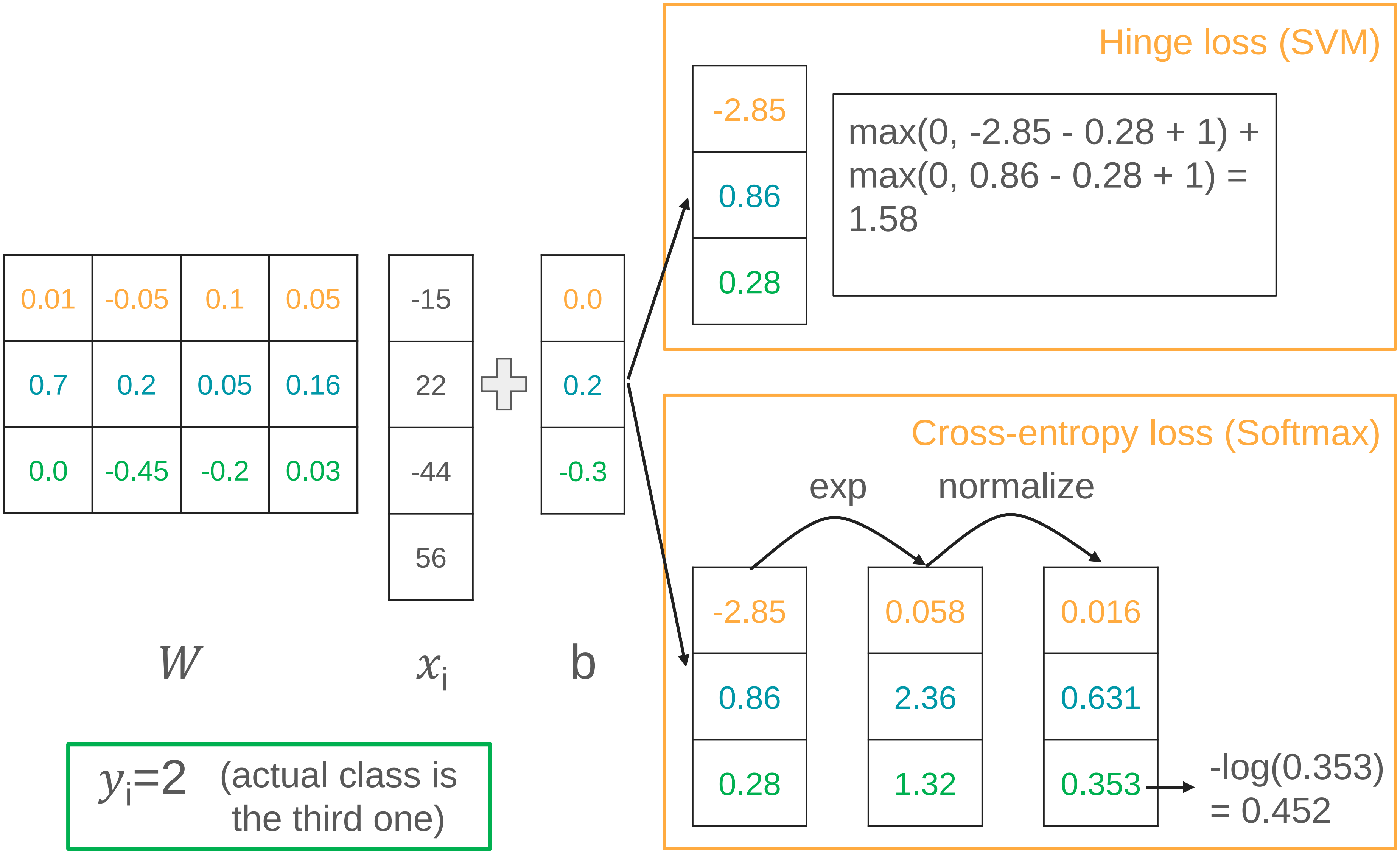
- What is the min/max possible loss?

- Usually at initialization W are small numbers, so all s ~= 0.
  What is the loss?

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

	unnormalized log probabilities	$\xrightarrow{\text{exp}}$	unnormalized probabilities	$\xrightarrow{\text{normalize}}$	normalized probabilities	
cat	**3.2**		**24.5**		**0.13**	$\longrightarrow$ L_i = -log(0.13) = 0.89
car	5.1		164.0		0.87	
frog	-1.7		0.18		0.00	

SVM vs. Softmax



SVM vs. Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

- Suppose we take a datapoint and changing its score slightly. What happens to the loss in both classifiers?

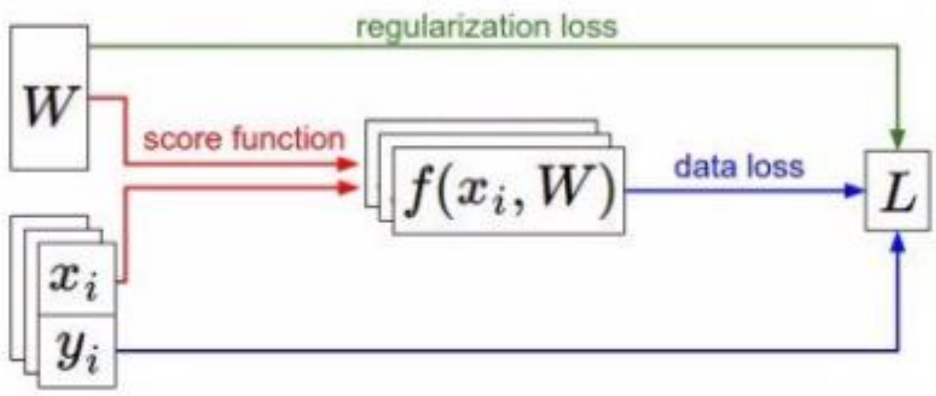
Assume scores:
[10, -2, 3]
[10, 9, 9]
[10, -100, -100]
Assume label $y_i = 0$ (first class)

Recap

- We have some dataset of (x,y)
- We have a score function: $s = f(x, W) = Wx$
- We have a loss function: $L_i + R(W)$ where:

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

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$ SVM



How to find the best W?

Optimization

Strategy 1: 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)
```

Optimization

Strategy 1: Random Search

- 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! (State-of-the-art is ~95%)

Optimization

Strategy 2: Follow down the slope

- In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In multiple dimensions, the gradient is the vector of (partial derivatives) along each direction.

Optimization

Strategy 2: Follow down the slope

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

	Current *W*	***W* + *h* (1st dimension)**	**Gradient *dW***	
	0.34	0.34 + 0.0001	-2.5	(1.25322 - 1.25347)/0.0001= -2.5
	-1.11	-1.11		
	0.78	0.78		
	0.12	0.12		
	0.55	0.55		
	2.81	2.81		
	-3.1	-3.1		
	-1.5	-1.5		
	0.33	0.33		
	...	...		
Loss	**1.25347**	**1.25322**		

Optimization

Strategy 2: Follow down the slope

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

	Current W	$W + h$ (2nd dimension)	Gradient dW	
	0.34	0.34	-2.5	
	-1.11	-1.11 + 0.0001	0.6	(1.25353 - 1.25347)/0.0001= 0.6
	0.78	0.78		
	0.12	0.12		
	0.55	0.55		
	2.81	2.81		
	-3.1	-3.1		
	-1.5	-1.5		
	0.33	0.33		
	...	...		
Loss	**1.25347**	**1.25353**		

Optimization

Strategy 2: Follow down the slope

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

	Current W	$W + h$ (3rd dimension)	Gradient dW	
	0.34	0.34	-2.5	
	-1.11	-1.11	0.6	
	0.78	0.78 + 0.0001	0	(1.25347 - 1.25347)/0.0001= 0
	0.12	0.12		
	0.55	0.55		
	2.81	2.81		
	-3.1	-3.1		
	-1.5	-1.5		
	0.33	0.33		
	...	...		
Loss	**1.25347**	**1.25347**		

Optimization

Strategy 2: Follow down the slope

- The approach we used is called ‘numerical gradient’.
- It is slow, approximate but easy

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

Optimization

Strategy 2: Follow down the slope (analytic gradient)

- In fact, the loss is just a function of W.

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

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

- We want $\nabla_W L$, the derivate of loss w.r.t W
- Use calculus to compute an analytic gradient

⟶ Analytical gradient: Exact, fast, error-prone!

In practice, always use analytic gradient, but check implementation with numerical gradient. This is called a gradient check.

Optimization

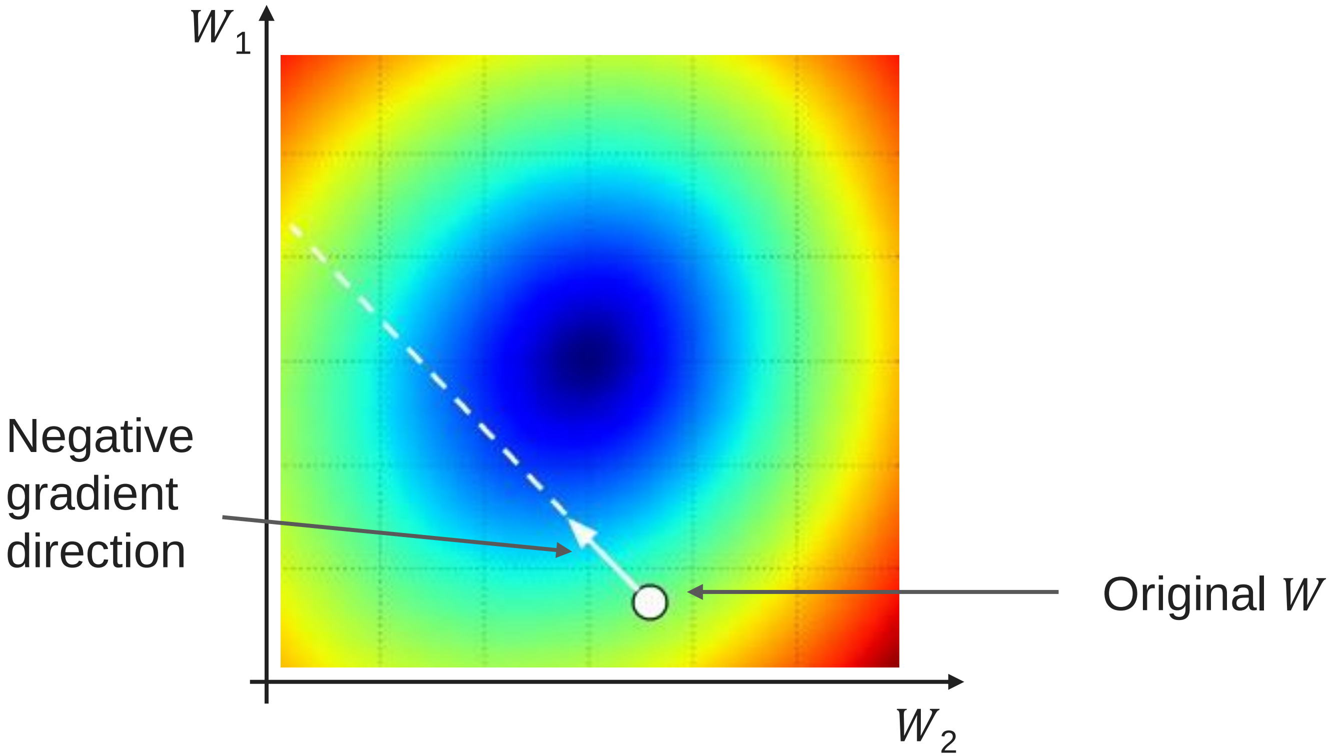
Strategy 2: Follow down the slope (analytic gradient)

Current W		**Gradient dW**
0.34		-2.5
-1.11		0.6
0.78	$\xrightarrow{\frac{\partial L}{\partial W}}$	0
0.12		0.2
0.55		0.7
2.81		-0.5
-3.1		1.1
-1.5		1.3
0.33		-2.1
...		…
Loss 1.25347		

Gradient Descent

```
# Vanilla Gradient Descent

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

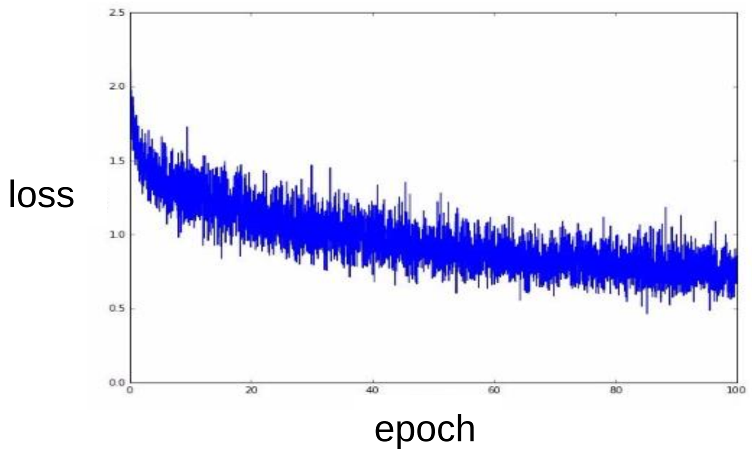


Mini-batch Gradient Descent

```
# 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
```

- Only use a small portion of the training set to compute the gradient.
- Common mini-batch sizes are 32/64/128 examples.
- When mini-batch contains only a single example, the process is called **Stochastic Gradient Descent (SGD)**



Example of optimization progress while training a neural network. (Loss over mini-batches goes down over time.)

Mini-batch 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
```

The effect of step size (or learning rate)