CS231n——Computer Vision

1. Course Introduction

Visual Data: Dark matter of the Internet

The field of computer vision is truly an interdisciplinary field, and it touches on many different areas of science and engineer and technology.

The history of vision
1600s the Renaissance period of time   camera obscura

1959   Hubel & Wiesel   the visual processing mechanism

1970   David Marr   Vision

- Input images: Perceived intensities
- Primal Sketch: Zero crossings, blobs, edges, bars, ends, virtual lines, groups, curves boundaries
- 2.5D Sketch: Local surface orientation and discontinuities in depth and in surface orientation
- 3-D Model Representation: 3-D models hierarchically organized in terms of surface and volumetric primitives

1979   Brooks & Binford   Generalized Cylinder

1973   Fischler and Elschlager   Pictorial Structure

- Object s composed of simple geometric primitives.

1987   David Lowe   Try to recognize razors by constructing lines and edges.

1997   Shi & Malik   Normalized Cut

1999   David Lowe   "SIFT" & Object Recognition

2001   Viola & Jones   Face Detection

2006   Lazebnik, Schmid & Ponce   Spatial Pyramid Matching

2005   Dalal& Trigas   Histogram of Gradients

2009   Felzenswalb, McAllester, Ramanan   Deformable Part Model

CS231-n overview

CS231-n focuses on one of the most important problems of visual recognition - image classification. There is a number of visual recognition problems that are related to image classification, such as object detection, image captioning.

Convolutional Neural Networks (CNN) have become an important tool for object recognition.

2.Image Classification

Image Classification: A core task in Computer Vision

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

The problem and challenge

- Semantic gap
- Viewpoint variation
- Illumination
- Deformation
- Occlusion
- Background Clutter
- Interclass variation

An image classifier

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

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

We want to come up with some algorithm or some method for these recognition tasks, which scales much more naturally to all the variety of objects in the world.

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

```
1  def train(image, labels):
2      #MAchine learning!
3      return model
4
5  def predict(model, test_images):
6      #Use model to predect labels
7      return test_labels
```

First classifier: Nearest Neighbor

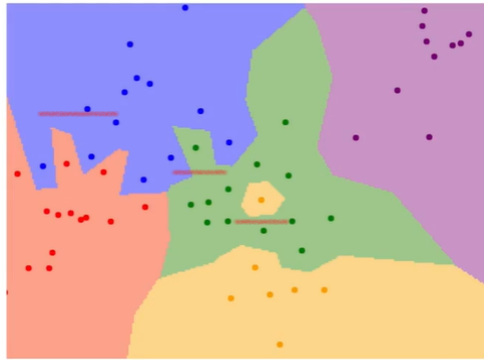1. During the training, it just need to memorize all the data and labels.

2. And then predict the label of the most similar training image.

Use the distance metric to compare images.
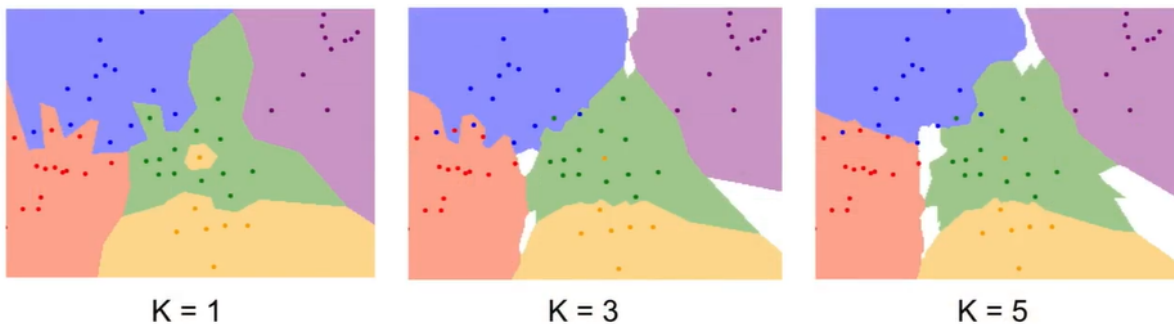
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

With N examples, the training is $O(1)$ and predicting is $O(N)$. This is bad, because we want classifiers that are fast at prediction; slow for training is ok.

the decision region

K-nearest neighbors

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

The K can smooth our boundaries and lead to better results.

Distance Metric:

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

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

Different distance metrics make different assumptions about the underlying geometry or topology that you'd expect in the space.

Each points in the two figures have the same distances to the original point. The first one use L1 distance and the second one using L2 distance.
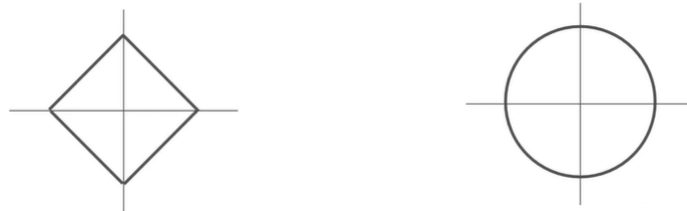
The L1 distance depends on the choice of your coordinate frame. So if you rotate the coordinate frame that would actually change the L1 distance between the points. Whereas change the frame that in the L2 distance doesn't matter.

You can use this algorithm on many different types of data.

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_P |I_1^P - I_2^P|$$

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_P (I_1^P - I_2^P)^2}$$


K = 1

K = 1

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. This is very problem-dependent because we must try them all out and see what works best.

Setting hyperparameters

Split data into train, validation and test, choose hyperparameters on validation and evaluate on test.

Split data into folds, try each fold as validation and average the results. This is useful for small datasets, but not used too frequently in deep learning.

Here is an example of 5-fold cross-validation for the value of K.
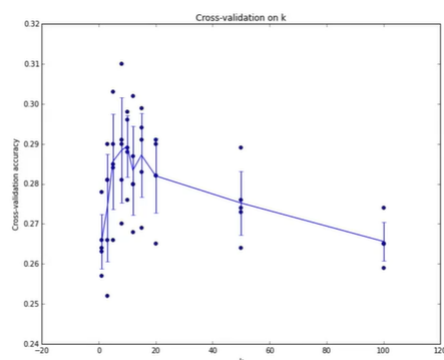
Each point is a single outcome, the line goes through the mean, and the bars indicated the standard deviation.

k-Nearest Neighbor on images never used.

- Very slow at test time.
- Distance metrics on pixels are not informative.
- Curse of dimensionality
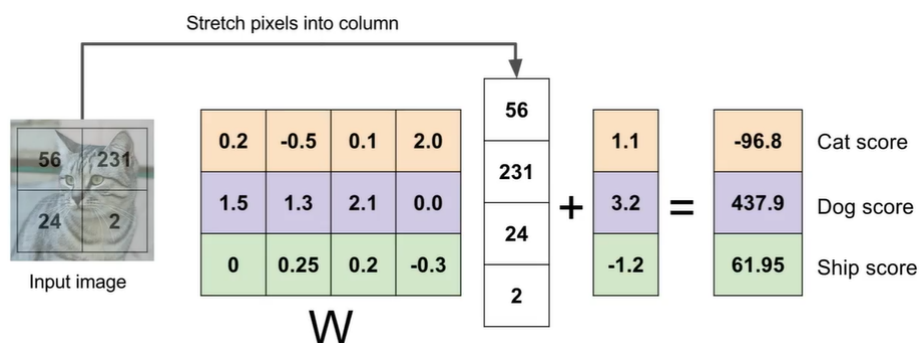
Linear Classification

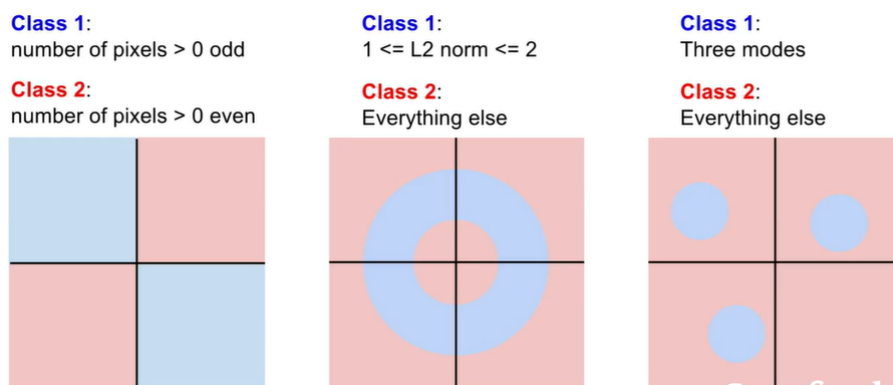Parametric Approach: Linear Classifier

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

X is the image input, such as an array of $32 \times 32 \times 3$ numbers (3072 in total)

W are parameters or weights

Output is 10 numbers giving class scores.

Hard cases for a linear classifier

So how to choose the right W?

3. Loss Functions and Optimization

Linear Classifier

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

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 label.

Loss over the dataset is a sum of loss over examples:

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

And we called **the multiclass SVM loss** the "Hinge loss".

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & if s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & otherwise \end{cases} = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

where y_1 is the category of the ground truth label for the example, so the s_{y_1} corresponds to the score of the true class for the i-th example in the training set.

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

The code to calculate the loss function using numpy.

```
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
```

Using **the multiclass SVM loss**, we have:

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

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

E.g. Suppose that we found a W such that L=0. Is this W unique?   No!

The concept of regularization

Model should be "simple", so it works on test data. We can add another term on the loss function which encourages the model to somehow pick a simpler W, where the concept of simple kind of depends on the task and the model.

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

- L_2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$
- L_1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic net $(L_1 + L_2)$ $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$
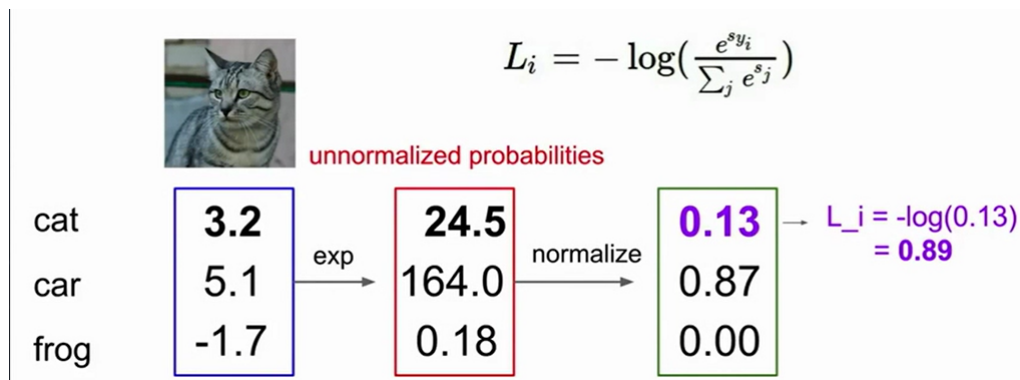
Softmax Classifier (Multinomial Logistic Regression)

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

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

Want tot maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

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

The only thing that the SVM loss cared about was getting that correct score to be greater than a margin above the incorrect scores. But now the softmax loss is actually quite different in this respect. The softmax loss actually always wants to drive that probability mass all the way to one. So the softmax class always try to continually to improve every single data point to get better and better.

Optimization

In practice, we tend to use various types of iterative methods, where we start with some solutions and then gradually improve it over time.

Random Research $\times$

Follow the slope

In one 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 dimension. The slope in any direction is the dot product of the direction with the gradient. The direction of steepest descent is the negative gradient.

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_{k} W_k^2$$
$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$
$$s = f(x; W) = Wx$$
$$\implies We\ want\ \nabla_W L$$

Numerical gradient: approximate, slow, easy to write

Analytic gradient: exact, fast, error-prone

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

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

The step size is a very significant super-parameter, which is often the first thing we try to set.

Stochastic Gradient Descent(SGD)

Full sum is expensive when N is large, so we often approximate sum using a minibatch of examples, and 32, 64, 128 (2^n) is common.

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

Image Features
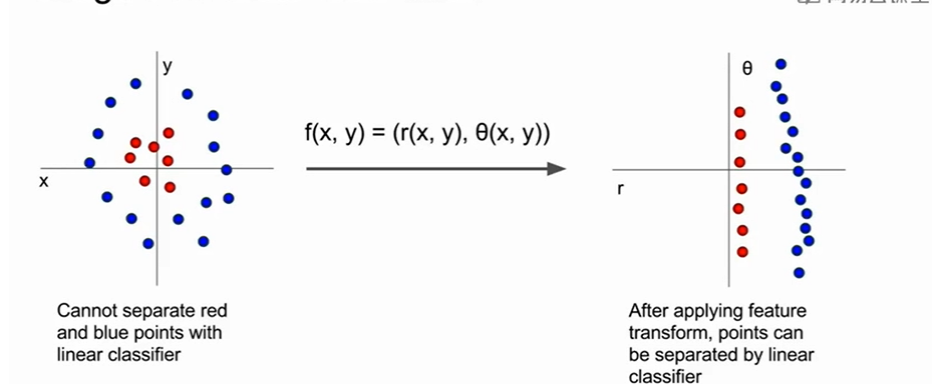
Example: Histogram of Oriented Gradients (HoG)

Example: Bag of words