Lecture 3 Loss function and optimization

Loss function

损失函数的主要定义是定量地去判断什么样的权重是好的，什么样的权重是坏的

A **loss function** tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and y_i is (integer) label

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

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

上图展示的是一个损失函数的基本定义方式，主要是通过某些函数来量化**计算值**与**Label**之间的差距。

SVM loss function —— hinge loss function

Multiclass SVM loss:

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the
scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

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

这是多分类的SVM loss function的计算过程，其中s_j是非正确分类的其他类的算出来的值，然后s_{yi}是正确分类的算出来的值。然后如果非正确分类的值加一小于正确分类的值，那么就可以认为是没有loss，反之，则认为存在着loss不可以。

上图中的1并不是问题的关键，我们主要做的是确定正确分类远远大于错误分类的值就可以了，至于是否是1，其实在计算的过程中这个常数是不影响的，因为他会随着权重的大小变化而变化，只是一个参考值而已。

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**
Losses:	2.9		

Multiclass SVM loss:

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

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

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**
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

Loss over full dataset is average:

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

L = (2.9 + 0 + 12.9)/3
= **5.27**

下图表示loss 函数的一个基本的计算方法。

下面是对损失函数提出的几个问题：

Q1: What happens to loss if car classes change a bit?

不会有什么变化，因为car分类的值远大于其他类的值，一点点的改变不影响结果。

Q2: What is the min/max possible loss?

最小值是0，最大值是正无穷。

Q3: At initialization W is small so all s are around zero. What is the loss?

loss最后会等于1，计算问题。 **这个问题会用于很多对代码的调试和判断对错中**

Q4: What if the sum was over all classes(including $j=y_i$)?

最后的损失函数会加1。

Q5: What if we used mean(average) instead of sum?

并不会影响什么，因为我们并不关心loss的真实值，我们关心的是相对值，所以任何缩放或者扩展的操作都不会影响到结果。

Q6: What if we used the loss function like this?

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

这就完全不同了，因为这个一个非线性的损失函数，其实就与原来的损失函数完全不一样的损失函数了。这主要取决于我们面对的问题。

```
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
```

这是用numpy构建hinge loss的一段代码。

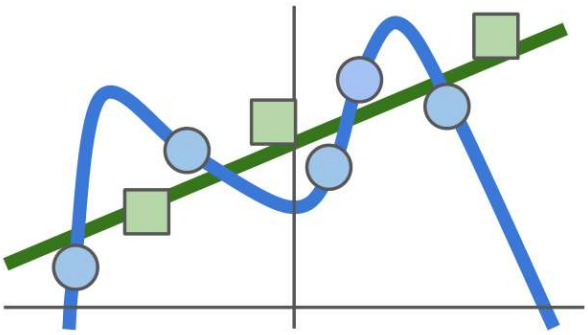
Suppose that we found a W such that L = 0. Is this W unique?

显然这个权重值并不是唯一的，因为这个loss只看重的是相对值，所以n倍的W的L仍然会等于0。

Regularization——损失函数的正则化

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data



Regularization: Model should be "simple", so it works on test data

Occam's Razor:
"Among competing hypotheses, the simplest is the best"
William of Ockham, 1285 - 1347

首先逐步解释一下上图的信息：

- 蓝色的点和线意味着在训练的过程中拟合的结果，是一个类似于高幂次的情况。
- 绿色的点是新加入的test data，绿色的线则是在考虑test data之后拟合出来的训练集的拟合曲线，与原拟合曲线相比存在着降维的操作。
- 为此，在原损失函数的基础上加入了正则化部分，用于使得模型更加的简单，类似于降维的操作。

几种正则化的方式

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

Max norm regularization (might see later)

Dropout (will see later)

Fancier: Batch normalization, stochastic depth

最常规使用的是L2正则化。

$$x = [1, 1, 1, 1] \qquad R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

其中可以看到，在L2正则化下，w_2有更小的范数，因此有更小的R(W)，所以在此正则化下可以认为w_2是更好的权重。

Softmax Classifier

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x_i; W)$$

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

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

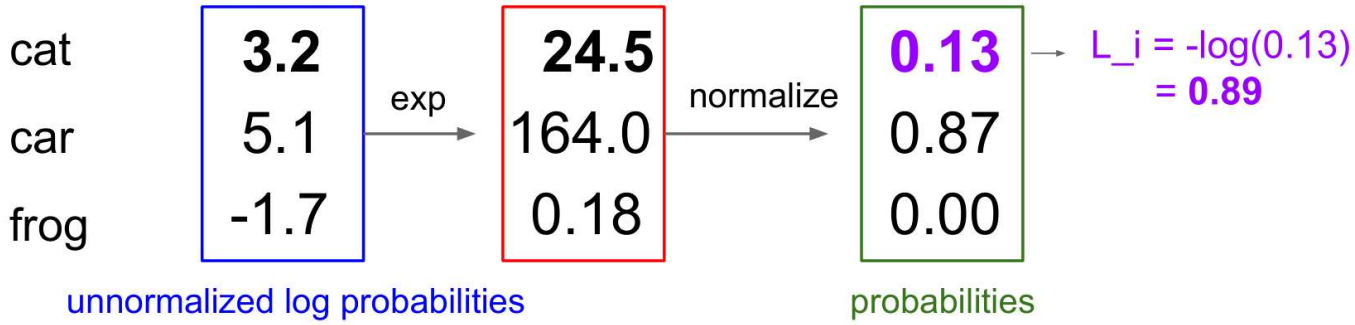
---

in summary: $$L_i = -\log(\frac{e^{sy_i}}{\sum_j e^{s_j}})$$

其实，softmax的基本思想是，让可能性最大的项占据尽可能多的概率，这样就会让分类更加的显然。

负号是因为损失函数，表示的应当是损失量，所以大体上应该是负值。

下面是一个例子，便于理解：



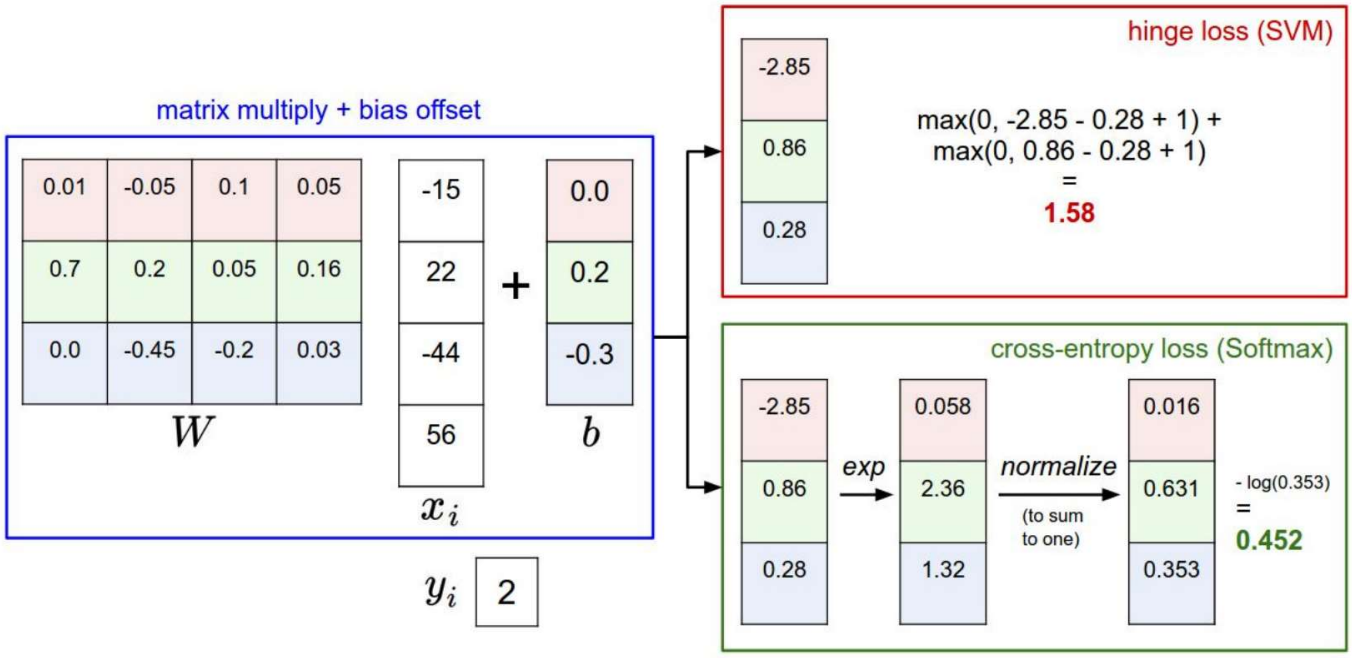
Q1: What is the min/max possible loss L_i

最小值是0，最大值是正无穷，虽然在现实操作之中你永远得不到正无穷。

Q2: Usually at initialization W is small so all s are around zero. What is the loss?

答案是-log(1/n)，其中n是总的类数。这就在初始化阶段给我们判断模型是否正确的方式了。比如我们在第一次迭代的时候看softmax的损失值，如果损失值与-log(1/n)相差甚远，那么基本断定你这个模型有很大的问题了。

两种损失函数的比较



从上图和两者的定义之中我们其实可以看出来，在定量表示损失的过程中，SVM考量的更多的是错的程度，而Softmax更多的是考量对的程度。所以其实整体上也是比较好区分的。

当然，当我们对正确的情况做改动，是之变得更加准确的时候，SVM其实并不会特别明显的表示出来这样的变化，但是Softmax则会在概率上表现出正确选项的加强。

Optimization

这里主要解决的问题是：**如何更快更好的得到最优的权重**?

Strategy#1: A first very bad idea solution: *Random Search*

设定一个尝试次数，疯狂的选出随机数然后尝试，最后尝试完了得出尝试过程中的最优解。**碰运气，很垃圾**

Strategy#2: Follow the slope——斜率或者*梯度*

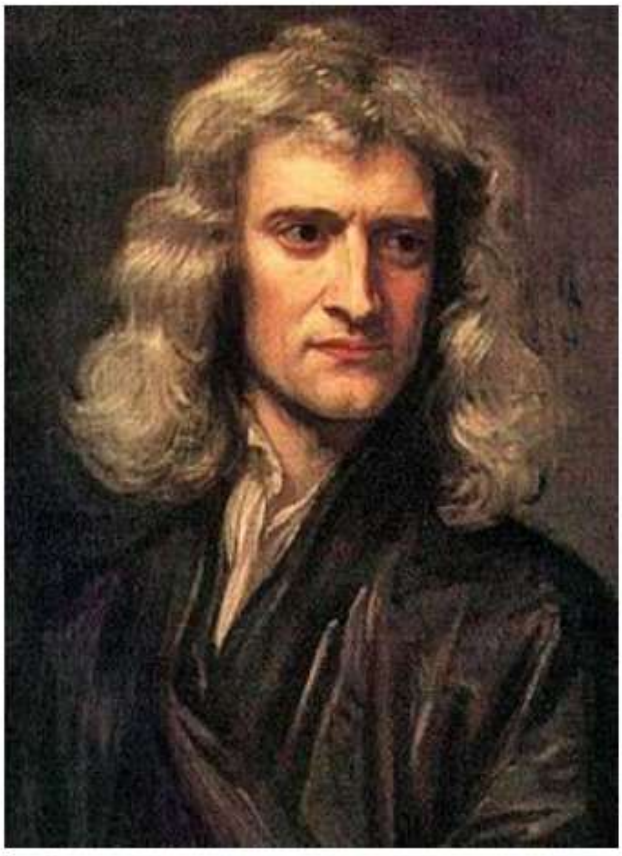
current W:	W + h (third dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[0.34, -1.11, 0.78 + **0.0001**, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[-2.5, 0.6, **0**, ?, ?,…]
loss 1.25347	loss 1.25347	

(1.25347 - 1.25347)/0.0001
= 0

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

计算机里面计算梯度的方式很常规的就是极限逼近的方式，但是可以想到，在W的规模较小的情况下是没啥大问题的，但是在实际情况下W的规模是很大的，这样就存在着一个问题就是——**计算量过大，计算代价无法接受。**

当然在受益于微积分的鼻祖——下面这两位大哥，一维的梯度在明确损失函数的情况下是可以直接写出关系式来的。这样是比较好的一种情况。



Gradient Descent —— 梯度下降

```
# Vanilla Gradient Descent

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

梯度下降是一个非常高效的选择最佳W的方式，因为对于一个比较正常的函数，寻求最小值的最快方式就是按照梯度下降最快的方向一路跑就到了。

上面是梯度下降的一段代码，可以看到，其中最关键的量莫过于**step_size**了，这个量简单来说就是步长，在大部分时候被称作另外一个名字——**学习率**——**learning rate**，这个数是可以说深度学习里面最为重要的超参了。之后的学习中，找这个超参的最优值可以说是debug过程中最为重要的一步。

Stochastic Gradient Descent(SDG) -- 随机梯度下降

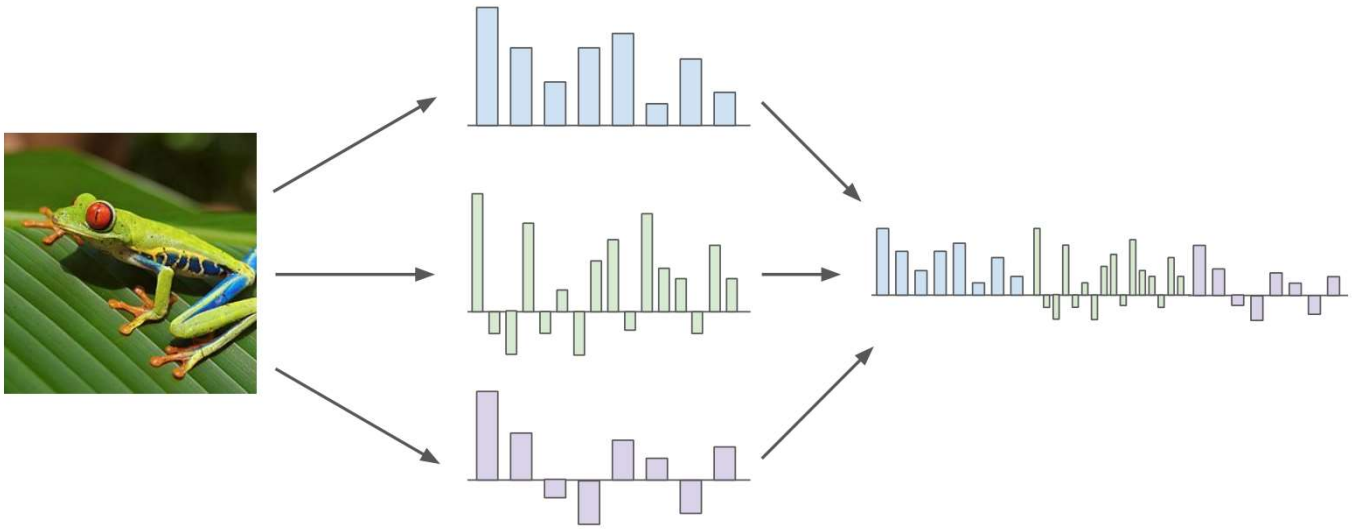
```
while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

如果我们用正规的梯度下降的话，每次我们需要将整个训练集迭代，这样的话对于大数据集，计算的代价是非常大的。所以我们选择一种新的算法——**随机梯度下降**，也就是每次我们用于计算梯度的样本是一小部分的数据集，而并不是整个的数据集，这样大大降低计算量。基本的代码如上图所示。其中，**minibatch**的大小一般选择的是2的倍数，也就是32,64,128之类的。

下面这个link是他们做的演示梯度下降各种损失函数和学习率是如何训练的一个网页，需要翻墙：

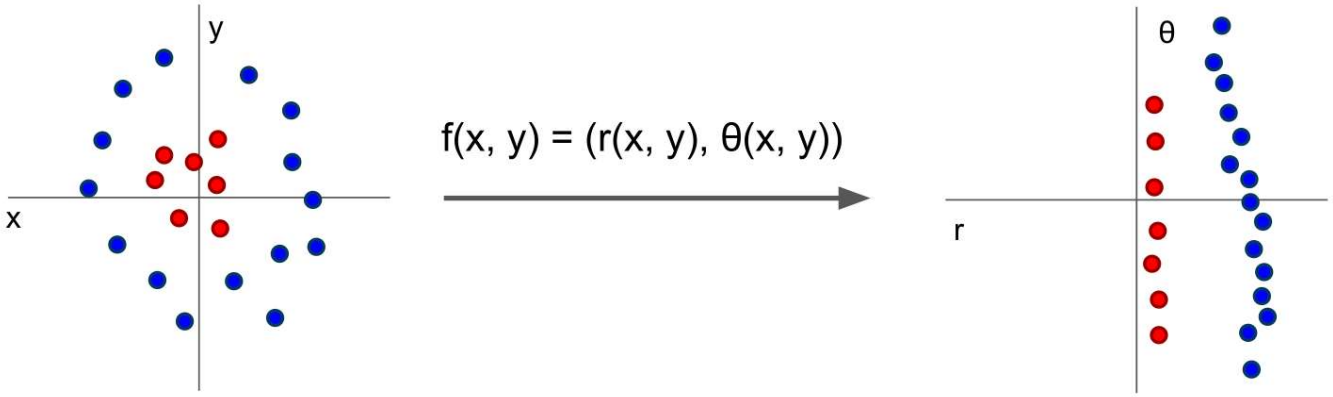
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Image Features



对于图像分类来说，直接将原图的像素导入到线性分类器中是不合理而且效果不好的，所以我们需要将图像进行处理，得到**Image Features**，再将这些特征组合得到分类器的最终输入。

比如一个非常简单的例子，左图中的元素是没有办法通过线性分类器做分类的，因为他们相当于是圆环分布，但是在做一个极坐标处理之后，重新导入线性分类器，分类就变得非常简单了。

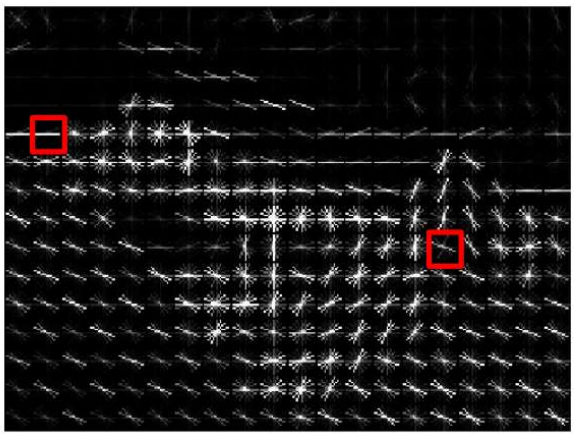


Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier

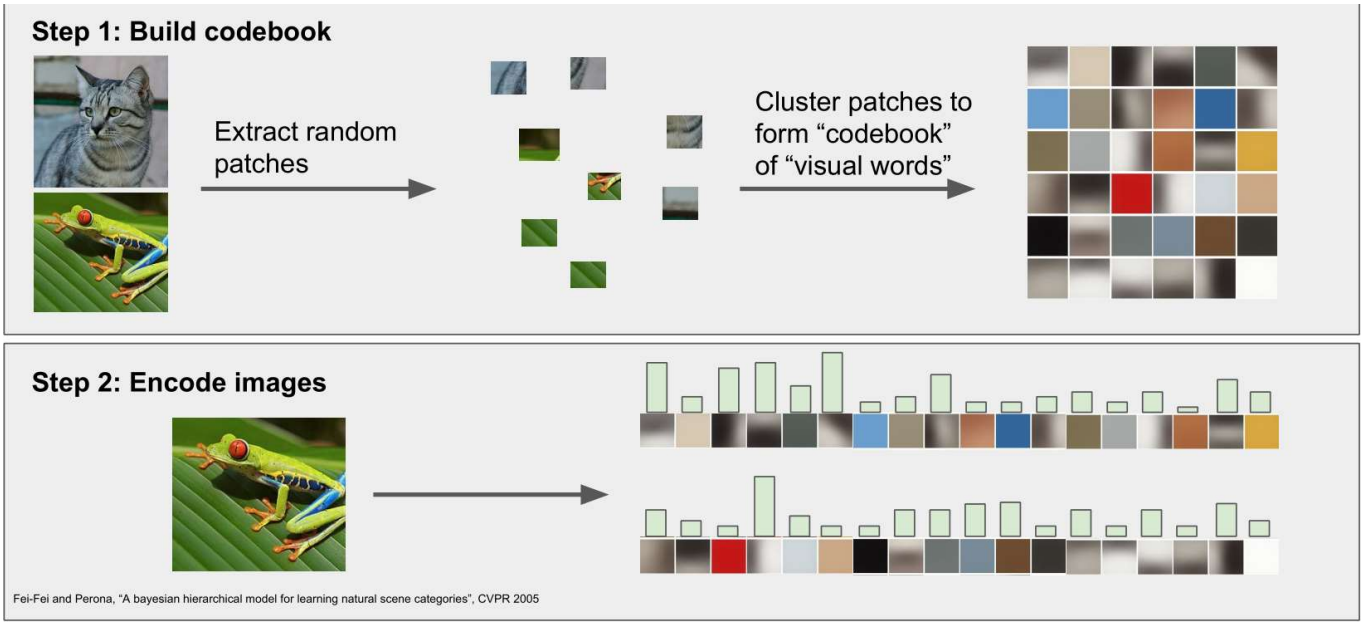
第一个例子就是**有向梯度直方图**，也就是下面右边的那个图。他就是将青蛙图中的边缘部分做了有向的描述。

Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins



Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

第二个例子就是**词袋**，这个概念是从NLP那边传过来的，用于检测每一张图中出现某些特定特征的图像的次数，用于解决图像的分类。



下面就是原本基于CV知识的非卷积神经网络而基于图像特征的图像分类与卷积神经网络分类的步骤区别：(用于引出下一章对卷积神经网络的介绍)