

# assignment1 -Q2 Multiclass Support Vector Machine

- |    |   |
|----|---|
| 1. | 编写：郭承坤 观自在降魔 <a href="#">Fanli SlyneD</a> |
| 2. | 校对：毛丽                                     |
| 3. | 总校对与审核：寒小阳                                |

代码环境：**python3.6.1(anaconda4.4.0) && ubuntu16.04** 测试通过

## 任务

- 完成一个基于SVM的全向量化**损失函数**
- 完成**解析梯度**的全向量化表示
- 用数值梯度来**验证你的实现**
- 使用一个验证集去**调优学习率和正则化强度**
- 运用**随机梯度下降**去优化损失函数
- 可视化**最后的学习得到的权重

## 线性SVM分类器

可以简单地认为，线性分类器给样本分类，每一个可能类一个分数，正确的分类分数，应该比错误的分类分数大。为了使分类器在分类未知样本的时候，鲁棒性更好一点，我们希望正确分类的分数比错误分类分数大得多一点。所以我们设置一个阈值 $\Delta$ ，让正确分类的分数至少比错误分数大 $\Delta$ ，这是我们期望的安全距离。这就得到了hinge损失函数。

$$L_i = \sum_{j \neq y_i} \max(0, S_j - S_{y_i} + \Delta)$$

其中， $L_i$  表示第*i*个样本的loss函数。 $S_{y_i}$  表示第*i*个样本正确分类的标签的分数， $S_j$ 表示第*i*个样本对应某个标签的分数。当我们把线性方程  $f(x) = w^T x + b$  的系数乘以一个倍数之后，得分将全部乘以这个倍数。所以 $\Delta$ 的值具体多少是没有意义的，有意义的是它相对于其他类上得分的大小，所以这里可以直接设定为1。如果想获得更专业的数学表达，可以查看周志华老师的《机器学习》等书籍。

然后再加入一个正则项来衡量模型的复杂度，学过线性回归的同学自然想到正则化的L1或者L2正则，比如 $\frac{1}{2}||w||^2$ ，然后目标函数就变成：

$$L_i = \sum_{j \neq y_i} \max(0, S_j - S_{y_i} + \Delta) + \frac{\lambda}{2} ||w||^2$$

到了这一步，基本上能看出一些与传统机器学习推导SVM公式的思路异曲同工的地方了，至于求解，用SGD就可以了。

以上，就是本次多分类SVM的一些心得，要细究起来，水还是很深的。

## ———开始写作业的分割线————

### notebook的一些预备代码

```
1. import random
2. import numpy as np
3. from cs231n.data_utils import load_CIFAR10
4. import matplotlib.pyplot as plt
5.
6. from __future__ import print_function
7.
8. #基本设定 让matplotlib画的图出现在notebook页面上，而不是新建一个画图窗口。
9. %matplotlib inline
10. plt.rcParams['figure.figsize'] = (10.0, 8.0) # 设置默认的绘图窗口大小
11. plt.rcParams['image.interpolation'] = 'nearest'
12. plt.rcParams['image.cmap'] = 'gray'
13.
14. # 另一个设定 可以使notebook自动重载外部python 模块. [点击此处查看详情] [4]
15. # 也就是说，当从外部文件引入的函数被修改之后，在notebook中调用这个函数，得到的被改
16. %load_ext autoreload
17. %autoreload 2
```

### 加载 CIFAR-10 原始数据

```
1. # 这里加载数据的代码在 data_utils.py 中，会将data_batch_1到5的数据作为训练集,
2. cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
3. X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)
4.
5. # 为了对数据有一个认识，打印出训练集和测试集的大小
6. print('Training data shape: ', X_train.shape)
7. print('Training labels shape: ', y_train.shape)
8. print('Test data shape: ', X_test.shape)
9. print('Test labels shape: ', y_test.shape)
```

输出：

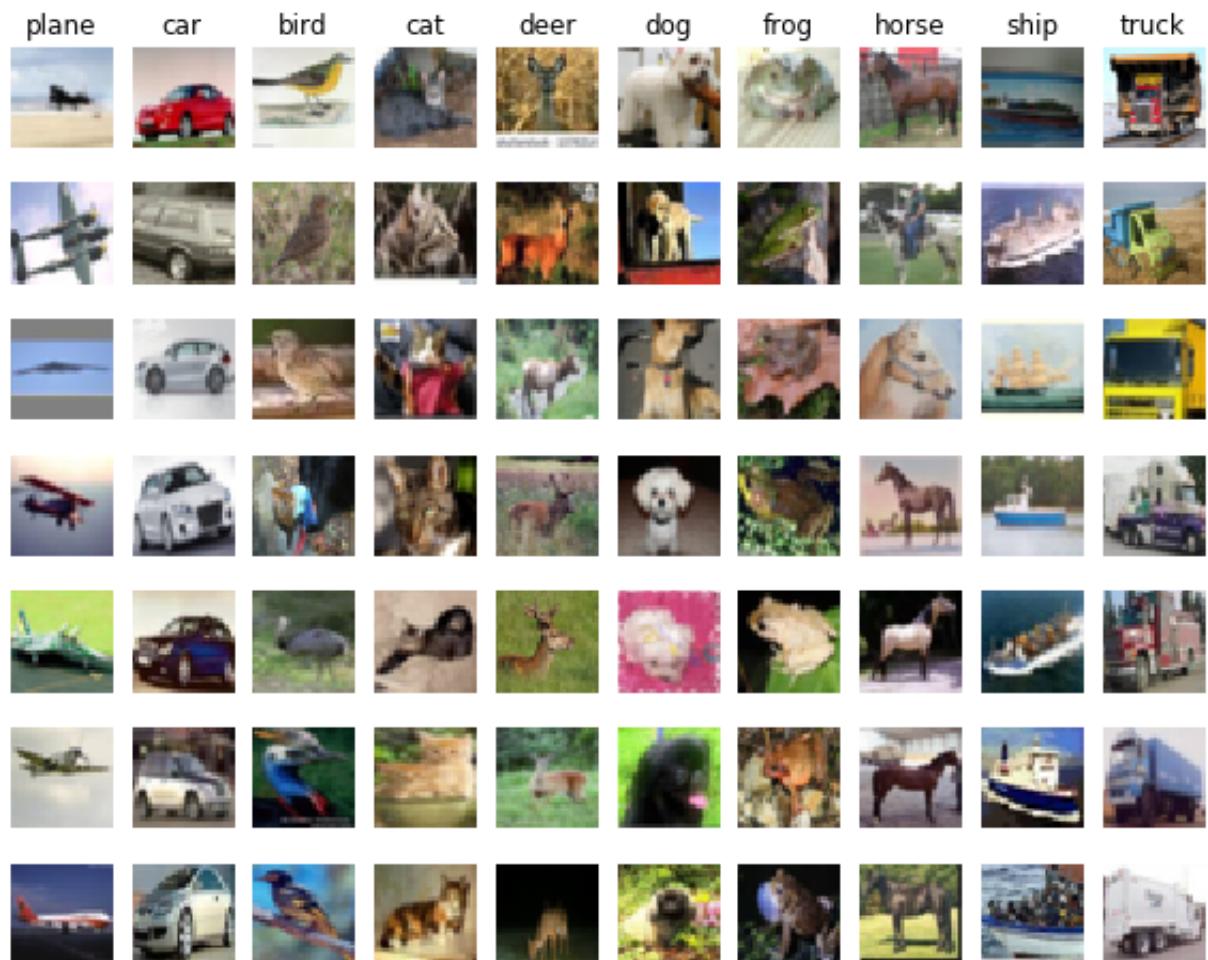
```
1. Training data shape: (50000, 32, 32, 3)
2. Training labels shape: (50000,)
3. Test data shape: (10000, 32, 32, 3)
4. Test labels shape: (10000,)
```

## 看看数据集中的样本

这里我们将训练集中每一类的样本都随机挑出几个进行展示

```
1. classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
2. num_classes = len(classes)
3. samples_per_class = 7
4. for y, cls in enumerate(classes):
5.     idxs = np.flatnonzero(y_train == y)
6.     idxs = np.random.choice(idxs, samples_per_class, replace=False)
7.     for i, idx in enumerate(idxs):
8.         plt_idx = i * num_classes + y + 1
9.         plt.subplot(samples_per_class, num_classes, plt_idx)
10.        plt.imshow(X_train[idx].astype('uint8'))
11.        plt.axis('off')
12.        if i == 0:
13.            plt.title(cls)
14. plt.show()
```

输出



将数据分割为训练集，验证集和测试集。

另外我们将创建一个“开发集”作为训练集的子集，算法开发时可以使用这个开发集，使我们的代码运行更快。

个人看法，这个development set在这个note book里，就是从训练集中进行了一个小的抽样，用来测试一些结论。

在吴恩达老师的深度学习课程和其他一些书籍里，development set 应该是被等同于验证集的，这里说明下。

```
1. num_training = 49000
2. num_validation = 1000
3. num_test = 1000
4. num_dev = 500
5.
6. # 验证集将会是从原始的训练集中分割出来的长度为 num_validation 的数据样本点
7. mask = range(num_training, num_training + num_validation)
8. X_val = X_train[mask]
9. y_val = y_train[mask]
10.
11. # 训练集是原始的训练集中前 num_train 个样本
12. mask = range(num_training)
13. X_train = X_train[mask]
14. y_train = y_train[mask]
15.
16. # 我们也可以从训练集中随机抽取一小部分的数据点作为开发集
17. mask = np.random.choice(num_training, num_dev, replace=False)
18. X_dev = X_train[mask]
19. y_dev = y_train[mask]
20.
21. # 使用前 num_test 个测试集点作为测试集
22. mask = range(num_test)
23. X_test = X_test[mask]
24. y_test = y_test[mask]
25.
26. print('Train data shape: ', X_train.shape)
27. print('Train labels shape: ', y_train.shape)
28. print('Validation data shape: ', X_val.shape)
29. print('Validation labels shape: ', y_val.shape)
30. print('Test data shape: ', X_test.shape)
31. print('Test labels shape: ', y_test.shape)
```

输出为

```
Train data shape: (49000, 32, 32, 3)
Train labels shape: (49000,)
Validation data shape: (1000, 32, 32, 3)
Validation labels shape: (1000,)
Test data shape: (1000, 32, 32, 3)
Test labels shape: (1000,)
```

## 数据预处理，将原始数据转成二维数据

```
1. #np.reshape(input_array, (k,-1)), 其中k为除了最后一维的维数, -1表示并不人为指  
2. #将所有样本, 各自拉成一个行向量, 所构成的二维矩阵, 每一行就是一个样本, 即一行有32X3  
3.  
4. X_train = np.reshape(X_train, (X_train.shape[0], -1))  
5. X_val = np.reshape(X_val, (X_val.shape[0], -1))  
6. X_test = np.reshape(X_test, (X_test.shape[0], -1))  
7. X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))  
8.  
9. # 很正常地, 要打印出来数据的形状看看。  
10. print('Training data shape: ', X_train.shape)  
11. print('Validation data shape: ', X_val.shape)  
12. print('Test data shape: ', X_test.shape)  
13. print('dev data shape: ', X_dev.shape)
```

输出

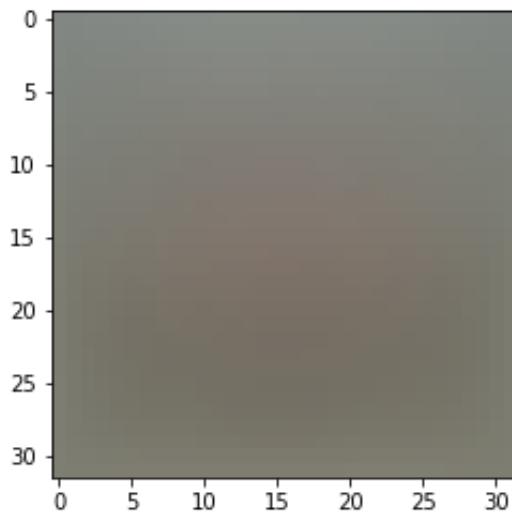
```
Training data shape: (49000, 3072)  
Validation data shape: (1000, 3072)  
Test data shape: (1000, 3072)  
dev data shape: (500, 3072)
```

## 预处理，减去图像的平均值

```
1. # 首先, 基于训练数据, 计算图像的平均值  
2. mean_image = np.mean(X_train, axis=0) #计算每一列特征的平均值, 共32x32x3个特  
3. print(mean_image.shape)  
4. print(mean_image[:10]) # 查看一下特征的数据  
5. plt.figure(figsize=(4,4)) #指定画图的框图大小  
6. plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # 将平均值可视化  
7. plt.show()
```

输出

```
(3072,)  
[ 130.64189796  135.98173469  132.47391837  130.05569388  135.34804082  
 131.75402041  130.96055102  136.14328571  132.47636735  131.48467347]
```



```
1. # 然后：训练集和测试集图像分别减去均值#
2. X_train -= mean_image
3. X_val -= mean_image
4. X_test -= mean_image
5. X_dev -= mean_image
```

```
1. # 最后，在X中添加一列1作为偏置维度，这样我们在优化时候只要考虑一个权重矩阵W就可以啦
2. X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
3. X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
4. X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
5. X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])
6.
7. print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
8. #看看各个数据集的shape，可以看出多了一列
```

输出

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

## SVM 分类器

这部分的code在 xxx/cs231n/classifiers/linear\_svm.py 里, 请按要求补充

我们已经实现了一个 `compute_loss_naive` 方法, 使用循环实现的loss计算.

```
1. # 评估我们提供给你的loss的朴素的实现.
2.
3. from cs231n.classifiers.linear_svm import svm_loss_naive
4. import time
5.
6. # 生成一个很小的SVM随机权重矩阵
7. # 真的很小, 先标准正态随机然后乘0.0001
8. W = np.random.randn(3073, 10) * 0.0001
9.
10. loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005) # 从dev数据集种的
11. print('loss: %f' % (loss, ))
```

输出

```
loss: 9.417096
```

上面返回的 `grad` 现在全是0. 推导并实现SVM损失函数, 写在 `svm_loss_native` 方法中.

为了检验你是否已经正确地实现了梯度算法, 你可以用数值方法估算损失函数的梯度, 然后比较数值与你用方程计算的值, 我们在下方提供了代码。

## linear\_svm.py

```

1. def svm_loss_naive(W, X, y, reg):
2.     """
3.     使用循环实现的SVM loss 函数。
4.     输入维数为D，有C类，我们使用N个样本作为一批输入。
5.     输入：
6.     -W: 一个numpy array, 形状为 (D, C) , 存储权重。
7.     -X: 一个numpy array, 形状为 (N, D), 存储一个小批数据。
8.     -y: 一个numpy array, 形状为 (N,)， 存储训练标签。y[i]=c 表示 x[i]的标签为c。
9.     -reg: float, 正则化强度。
10.
11.    输出一个tuple:
12.        - 一个存储为float的loss
13.        - 权重W的梯度，和W大小相同的array
14.    """
15.    dW = np.zeros(W.shape) # 梯度全部初始化为0
16.    # 计算损失和梯度
17.    num_classes = W.shape[1]
18.    num_train = X.shape[0]
19.    loss = 0.0
20.    for i in range(num_train):
21.        scores = X[i].dot(W)
22.        correct_class_score = scores[y[i]]
23.        for j in range(num_classes):
24.            if j == y[i]:
25.                continue
26.            margin = scores[j] - correct_class_score + 1 # 记住 delta = 1
27.            if margin > 0:
28.                loss += margin
29.                dW[:, y[i]] += -X[i, :].T
30.                dW[:, j] += X[i, :].T
31.
32.    # 现在损失函数是所有训练样本的和。但是我们要的是它们的均值。所以我们用 num_train
33.
34.    loss /= num_train
35.    dW /= num_train
36.    # 加入正则项
37.    loss += reg * np.sum(W * W)
38.    dW += reg * W
39.
40. #####
41.    # 任务:
42.    # 计算损失函数的梯度并存储在dW中。 相比于先计算loss然后计算梯度,
43.    # 同时计算梯度和loss会更加简单。所以你可能需要修改上面的代码, 来计算梯度
44. #####
45.
46.    return loss, dW

```

## 验证梯度结果

回到notebook.

```

1. # 实现梯度之后, 运行下面的代码重新计算梯度.
2. # 输出是grad_check_sparse函数的结果, 2种情况下, 可以看出, 其实2种算法误差已经几乎
3. loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)
4.
5. # 对随机选的几个维度计算数值梯度, 并把它和你计算的解析梯度比较. 所有维度应该几乎相等
6. from cs231n.gradient_check import grad_check_sparse
7.
8.
9. f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
10. grad_numerical = grad_check_sparse(f, W, grad)
11.
12. # 再次验证梯度. 这次使用正则项. 你肯定没有忘记正则化梯度吧~
13. print('turn on reg')
14. loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
15. f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
16. grad_numerical = grad_check_sparse(f, W, grad)

```

输出:

```

numerical: -5.807950 analytic: -5.807950, relative error: 1.828298e-11
numerical: -14.928666 analytic: -14.928666, relative error: 9.989241e-12
numerical: 1.059552 analytic: 1.059552, relative error: 6.114175e-11
numerical: -29.481796 analytic: -29.481796, relative error: 5.863187e-13
numerical: 23.762741 analytic: 23.762741, relative error: 1.351804e-11
numerical: 8.872796 analytic: 8.872796, relative error: 2.925921e-11
numerical: -2.480627 analytic: -2.480627, relative error: 1.035958e-10
numerical: 7.650113 analytic: 7.589849, relative error: 3.954297e-03
numerical: 24.438469 analytic: 24.438469, relative error: 1.619239e-11
numerical: 0.781688 analytic: 0.781688, relative error: 6.861017e-11
turn on reg
numerical: 4.243960 analytic: 4.268399, relative error: 2.871033e-03
numerical: -22.515384 analytic: -22.510695, relative error: 1.041591e-04
numerical: 18.283080 analytic: 18.286599, relative error: 9.622179e-05
numerical: 0.401949 analytic: 0.348694, relative error: 7.094580e-02
numerical: 28.537163 analytic: 28.535275, relative error: 3.308435e-05
numerical: -22.447485 analytic: -22.448065, relative error: 1.291663e-05
numerical: 0.700087 analytic: 0.701799, relative error: 1.221305e-03
numerical: 5.775369 analytic: 5.774106, relative error: 1.093817e-04
numerical: -4.901730 analytic: -4.911853, relative error: 1.031485e-03
numerical: 9.014401 analytic: 9.011687, relative error: 1.505957e-04

```

## 随堂练习 1 :

偶尔会出现梯度验证时候, 某个维度不一致, 导致不一致的原因是什么呢? 这是我们要考虑的一个因素么? 梯度验证失败的简单例子是?

提示: SVM 损失函数没有被严格证明是可导的.

可以看上面的输出结果, turn on reg前有一个是3.954297e-03的loss明显变大.

## 答案

解析解和数值解的区别，数值解是用前后2个很小的随机尺度(比如0.00001)进行计算，当Loss不可导的，两者会出现差异。比如 $S_{y_i}$ 刚好比 $S_j$ 大1.

## 完成 svm\_loss\_vectorized 方法

现在先计算loss,等一下完成梯度

### **linear\_svm.py**

```

1. def svm_loss_vectorized(W, X, y, reg):
2.     """
3.         结构化的SVM损失函数，使用向量来实现.
4.         输入和输出和svm_loss_naive一致.
5.     """
6.     loss = 0.0
7.     dW = np.zeros(W.shape) # 初始化梯度为0
8.
9.     #####
10.    # 任务:
11.    # 实现结构化SVM损失函数的向量版本. 将损失存储在loss变量中.
12.    #####
13.
14.    scores = X.dot(W)
15.    num_classes = W.shape[1]
16.    num_train = X.shape[0]
17.
18.    scores_correct = scores[np.arange(num_train), y]
19.    scores_correct = np.reshape(scores_correct, (num_train, -1))
20.    margins = scores - scores_correct + 1
21.    margins = np.maximum(0,margins)
22.    margins[np.arange(num_train), y] = 0
23.    loss += np.sum(margins) / num_train
24.    loss += 0.5 * reg * np.sum(W * W)
25.    #####
26.    #                                     代码结束
27.    #####
28.
29.
30.
31.    #####
32.    # 任务:
33.    # 使用向量计算结构化SVM损失函数的梯度，把结果保存在 dW.
34.    # 提示：不一定从头计算梯度，可能重用计算loss时的一些中间结果会更简单.
35.    #####
36.
37.    margins[margins > 0] = 1
38.    row_sum = np.sum(margins, axis=1)           # 1 by N
39.    margins[np.arange(num_train), y] = -row_sum
40.    dW += np.dot(X.T, margins)/num_train + reg * W   # D by C
41.
42.    #####
43.    #                                     代码结束
44.    #####
45.    return loss, dW

```

[回到notebook.](#)

```

1. tic = time.time()
2. loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
3. toc = time.time()
4. print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))
5.
6. from cs231n.classifiers.linear_svm import svm_loss_vectorized
7. tic = time.time()
8. loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
9. toc = time.time()
10. print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc -
11.
12. # The losses should match but your vectorized implementation should be
13. print('difference: %f' % (loss_naive - loss_vectorized))

```

输出:

```

Naive loss: 9.417096e+00 computed in 0.121770s
Vectorized loss: 9.417096e+00 computed in 0.005394s
difference: 0.000000

```

```

1. # 使用向量来计算损失函数的梯度.
2. # 朴素方法和向量法的结果应该是一样的，但是向量法会更快一点.
3.
4. tic = time.time()
5. _, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
6. toc = time.time()
7. print('Naive loss and gradient: computed in %fs' % (toc - tic))
8.
9. tic = time.time()
10. _, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
11. toc = time.time()
12. print('Vectorized loss and gradient: computed in %fs' % (toc - tic))
13.
14. # The loss is a single number, so it is easy to compare the values com
15. # by the two implementations. The gradient on the other hand is a matr
16. # we use the Frobenius norm to compare them.
17. difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
18. print('difference: %f' % difference)

```

```

Naive loss and gradient: computed in 0.127225s
Vectorized loss and gradient: computed in 0.003415s
difference: 0.000000

```

## 随机梯度下降

我们已经向量化并且高效地表达了损失、梯度，而且我们的梯度是与梯度数值解相一致的。因此我们可以利用SGD来最小化损失了。

## **linear\_classifier.py**

在文件 linear\_classifier.py 里，完成SGD函数 LinearClassifier.train()

```
1. def train(self, X, y, learning_rate=1e-3, reg=1e-5, num_iters=100,
2.           batch_size=200, verbose=False):
3.     """
4.     Train this linear classifier using stochastic gradient descent.
5.     使用随机梯度下降来训练这个分类器。
6.     输入:
7.     -X:一个 numpy array, 形状为 (N, D), 存储训练数据。共N个训练数据, 每个训练数据
8.     -Y:一个 numpy array, 形状为(N,), 存储训练数据的标签。y[i]=c 表示 x[i]的标签为
9.     -learning rate: float, 优化的学习率。
10.    -reg: float, 正则化强度。
11.    -num_iters: integer, 优化时训练的步数。
12.    -batch_size: integer, 每一步使用的训练样本数。
13.    -verbose: boolean, 若为真, 优化时打印过程。
14.    输出:
15.    一个存储每次训练的损失函数值的list。
16.    """
17.
18.    num_train, dim = X.shape
19.    num_classes = np.max(y) + 1 # 假设y的值是0...K-1, 其中K是类别数量。
20.
21.    if self.W is None:
22.        # 简易初始化 W
23.        self.W = 0.001 * np.random.randn(dim, num_classes)
24.
25.    # 使用随机梯度下降优化W
26.    loss_history = []
27.    for it in xrange(num_iters):
28.        X_batch = None
29.        y_batch = None
30.
31.        #####
32.        # 任务:
33.        # 从训练集中采样batch_size个样本和对应的标签, 在这一轮梯度下降中使用。
34.        # 把数据存储在 X_batch 中, 把对应的标签存储在 y_batch 中。
35.        # 采样后, X_batch 的形状为 (dim, batch_size), y_batch的形状为(batch_size,
36.        #
37.        # 提示: 用 np.random.choice 来生成 indices。有放回采样的速度比无放回采
38.        # 样的速度要快。
39.        #####
40.        batch_inx = np.random.choice(num_train, batch_size)
41.        X_batch = X[batch_inx,:]
42.        y_batch = y[batch_inx]
43.        #####
44.        # 结束
45.        #####
46.
47.        # evaluate loss and gradient
48.        loss, grad = self.loss(X_batch, y_batch, reg)
49.        loss_history.append(loss)
50.
51.        # perform parameter update
52.        #####
53.        # TODO:
```

```
54.     # 使用梯度和学习率更新权重
55. #####
56.     self.W = self.W - learning_rate * grad
57. #####
58.     # 结束
59. #####
60.
61.     if verbose and it % 100 == 0:
62.         print 'iteration %d / %d: loss %f' % (it, num_iters, loss)
63.
64.     return loss_history
```

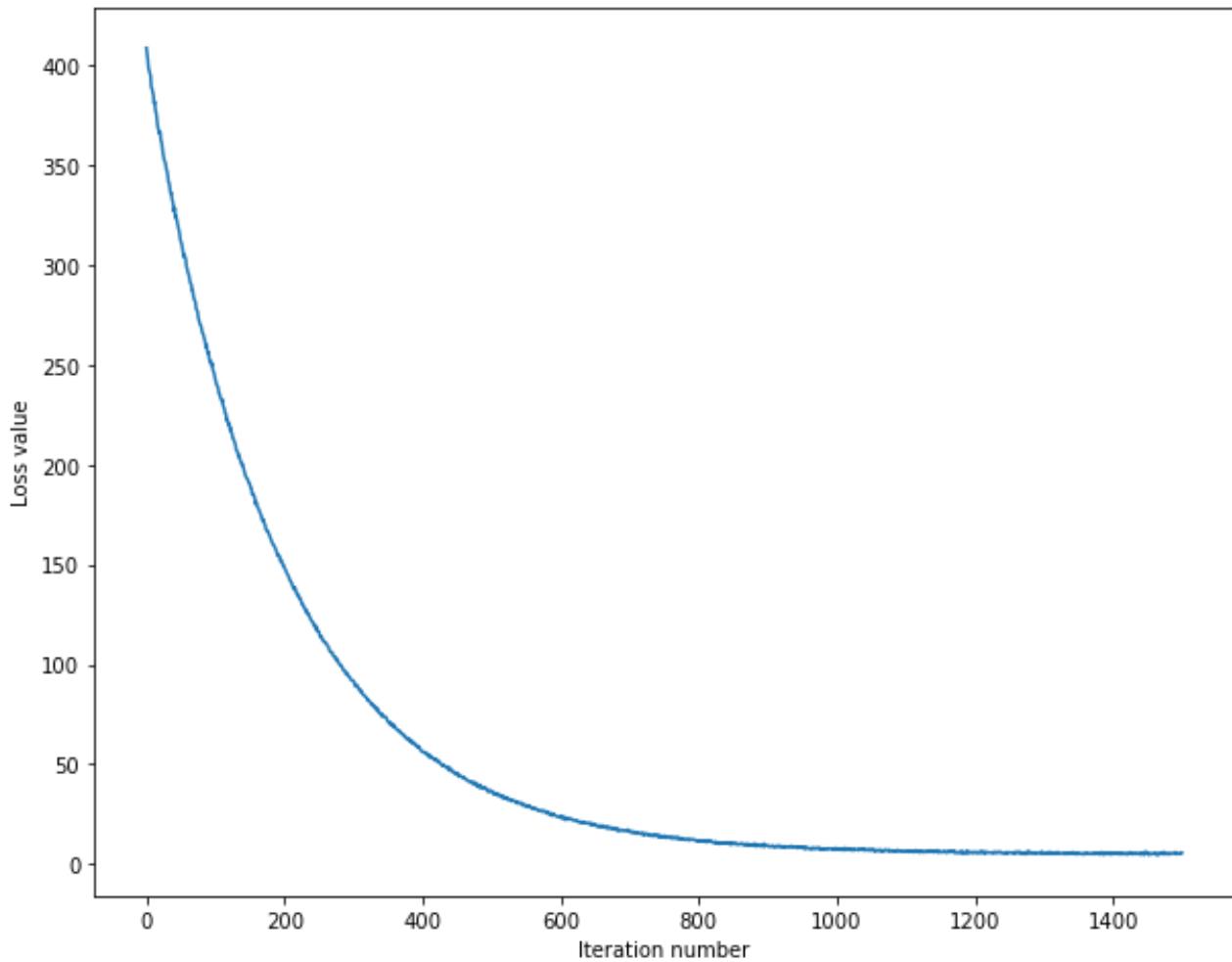
回到notebook。

```
1. from cs231n.classifiers import LinearSVM
2. svm = LinearSVM()
3. tic = time.time()
4. loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
5.                         num_iters=1500, verbose=True)
6. toc = time.time()
7. print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 406.551290
iteration 100 / 1500: loss 240.613943
iteration 200 / 1500: loss 146.111274
iteration 300 / 1500: loss 90.213982
iteration 400 / 1500: loss 56.328602
iteration 500 / 1500: loss 35.504821
iteration 600 / 1500: loss 23.727488
iteration 700 / 1500: loss 15.755706
iteration 800 / 1500: loss 11.425011
iteration 900 / 1500: loss 9.055450
iteration 1000 / 1500: loss 7.637563
iteration 1100 / 1500: loss 6.671778
iteration 1200 / 1500: loss 5.914076
iteration 1300 / 1500: loss 5.219254
iteration 1400 / 1500: loss 5.034812
That took 3.781817s
```

回到notebook。

```
1. # 大佬说了，有效的debug策略就是将损失和循环次数画出来。
2. plt.plot(loss_hist)
3. plt.xlabel('Iteration number')
4. plt.ylabel('Loss value')
5. plt.show()
```



```
1. # 编写函数 LinearSVM.predict, 评估训练集和验证集的表现。
2. y_train_pred = svm.predict(X_train)
3. print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
4. y_val_pred = svm.predict(X_val)
5. print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

输出

```
training accuracy: 0.380776
validation accuracy: 0.392000
```

## 使用验证集去调整超参数（正则化强度和学习率）

```

1. # 使用验证集去调整超参数（正则化强度和学习率），你要尝试各种不同的学习率
2. # 和正则化强度，如果你认真做，将会在验证集上得到一个分类准确度大约是0.4的结果。
3. # 设置学习率和正则化强度，多设几个靠谱的，可能会好一点。
4. # 可以尝试先用较大的步长搜索，再微调。
5.
6. learning_rates = [2e-7, 0.75e-7, 1.5e-7, 1.25e-7, 0.75e-7]
7. regularization_strengths = [3e4, 3.25e4, 3.5e4, 3.75e4, 4e4, 4.25e4, 4.
8.
9.
10. # 结果是一个词典，将形式为(learning_rate, regularization_strength) 的tuple
11.
12. results = {}
13. best_val = -1    # 出现的正确率最大值
14. best_svm = None # 达到正确率最大值的svm对象
15.
16. #####
17. # 任务：
18. # 写下你的code，通过验证集选择最佳超参数。对于每一个超参数的组合，
19. # 在训练集训练一个线性svm，在训练集和测试集上计算它的准确度，然后
20. # 在字典里存储这些值。另外，在 best_val 中存储最好的验证集准确度，
21. # 在best_svm中存储达到这个最佳值的svm对象。
22. #
23. # 提示：当你编写你的验证代码时，你应该使用较小的num_iters。这样SVM的训练模型
24. # 并不会花费太多的时间去训练。当你确认验证code可以正常运行之后，再用较大的
25. # num_iters 重跑验证代码。
26.
27. #####
28. for rate in learning_rates:
29.     for regular in regularization_strengths:
30.         svm = LinearSVM()
31.         svm.train(X_train, y_train, learning_rate=rate, reg=regular,
32.                   num_iters=1000)
33.         y_train_pred = svm.predict(X_train)
34.         accuracy_train = np.mean(y_train == y_train_pred)
35.         y_val_pred = svm.predict(X_val)
36.         accuracy_val = np.mean(y_val == y_val_pred)
37.         results[(rate, regular)]=(accuracy_train, accuracy_val)
38.         if (best_val < accuracy_val):
39.             best_val = accuracy_val
40.             best_svm = svm
41. #####
42. #                         结束
43. #####
44.
45. for lr, reg in sorted(results):
46.     train_accuracy, val_accuracy = results[(lr, reg)]
47.     print ('lr %e reg %e train accuracy: %f val accuracy: %f' % (lr, r
48.
49. print ('best validation accuracy achieved during cross-validation: %f'

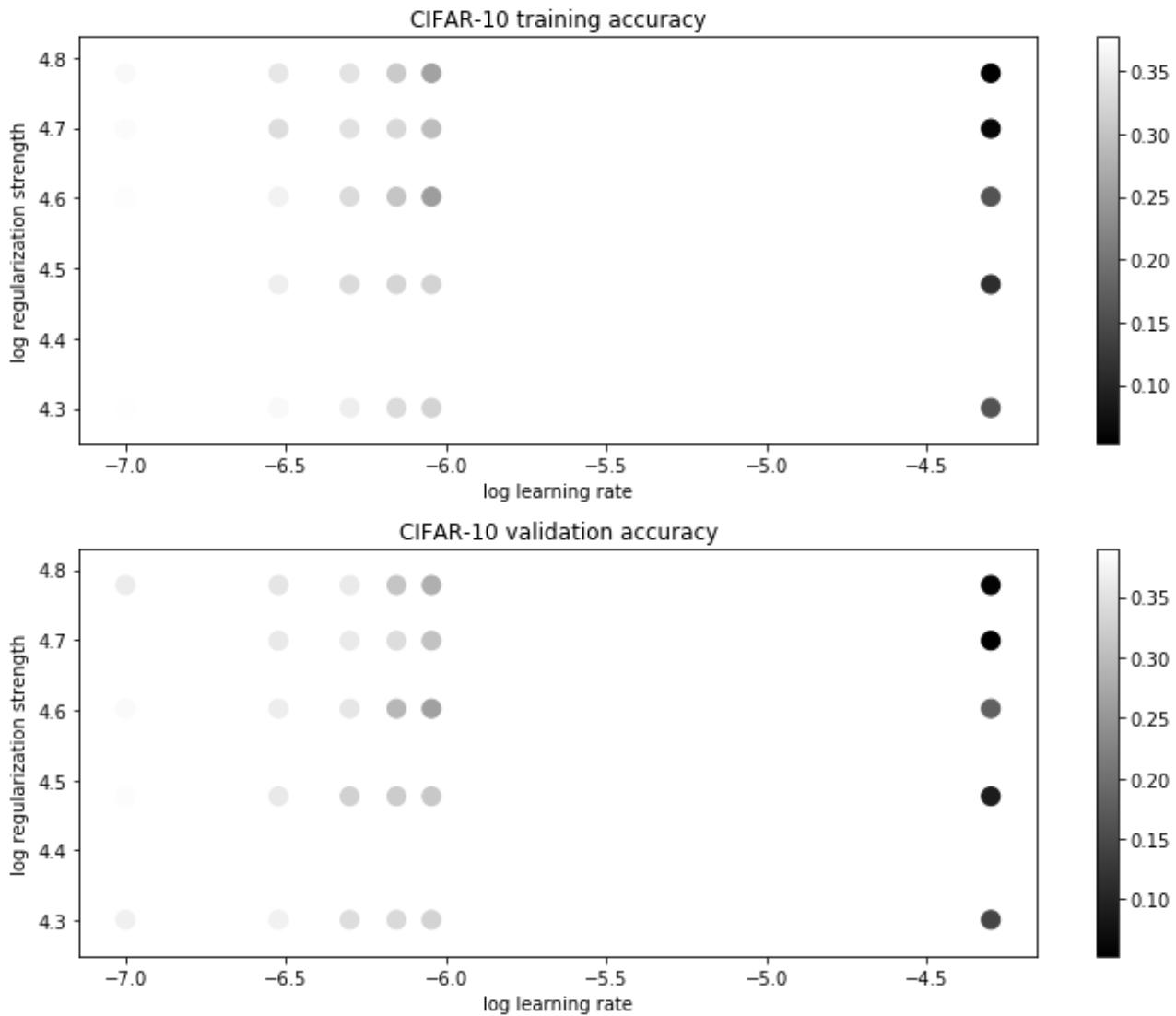
```

输出

```
lr 7.500000e-08 reg 3.000000e+04 train accuracy: 0.367980 val accuracy: 0.3
lr 7.500000e-08 reg 3.250000e+04 train accuracy: 0.374653 val accuracy: 0.3
lr 7.500000e-08 reg 3.500000e+04 train accuracy: 0.374980 val accuracy: 0.3
lr 7.500000e-08 reg 3.750000e+04 train accuracy: 0.374000 val accuracy: 0.3
lr 7.500000e-08 reg 4.000000e+04 train accuracy: 0.371694 val accuracy: 0.3
lr 7.500000e-08 reg 4.250000e+04 train accuracy: 0.374082 val accuracy: 0.3
lr 7.500000e-08 reg 4.500000e+04 train accuracy: 0.372673 val accuracy: 0.3
lr 7.500000e-08 reg 4.750000e+04 train accuracy: 0.374388 val accuracy: 0.3
lr 7.500000e-08 reg 5.000000e+04 train accuracy: 0.374367 val accuracy: 0.3
lr 1.250000e-07 reg 3.000000e+04 train accuracy: 0.374061 val accuracy: 0.3
lr 1.250000e-07 reg 3.250000e+04 train accuracy: 0.373041 val accuracy: 0.3
lr 1.250000e-07 reg 3.500000e+04 train accuracy: 0.373592 val accuracy: 0.3
lr 1.250000e-07 reg 3.750000e+04 train accuracy: 0.375327 val accuracy: 0.3
lr 1.250000e-07 reg 4.000000e+04 train accuracy: 0.371918 val accuracy: 0.3
lr 1.250000e-07 reg 4.250000e+04 train accuracy: 0.369347 val accuracy: 0.3
lr 1.250000e-07 reg 4.500000e+04 train accuracy: 0.370796 val accuracy: 0.3
lr 1.250000e-07 reg 4.750000e+04 train accuracy: 0.370041 val accuracy: 0.3
lr 1.250000e-07 reg 5.000000e+04 train accuracy: 0.367898 val accuracy: 0.3
lr 1.500000e-07 reg 3.000000e+04 train accuracy: 0.371592 val accuracy: 0.3
lr 1.500000e-07 reg 3.250000e+04 train accuracy: 0.373327 val accuracy: 0.4
lr 1.500000e-07 reg 3.500000e+04 train accuracy: 0.366429 val accuracy: 0.3
lr 1.500000e-07 reg 3.750000e+04 train accuracy: 0.371714 val accuracy: 0.3
lr 1.500000e-07 reg 4.000000e+04 train accuracy: 0.365327 val accuracy: 0.3
lr 1.500000e-07 reg 4.250000e+04 train accuracy: 0.366980 val accuracy: 0.3
lr 1.500000e-07 reg 4.500000e+04 train accuracy: 0.360286 val accuracy: 0.3
lr 1.500000e-07 reg 4.750000e+04 train accuracy: 0.362673 val accuracy: 0.3
lr 1.500000e-07 reg 5.000000e+04 train accuracy: 0.363429 val accuracy: 0.3
lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.375224 val accuracy: 0.3
lr 2.000000e-07 reg 3.250000e+04 train accuracy: 0.371286 val accuracy: 0.3
lr 2.000000e-07 reg 3.500000e+04 train accuracy: 0.368449 val accuracy: 0.3
lr 2.000000e-07 reg 3.750000e+04 train accuracy: 0.360633 val accuracy: 0.3
lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.360653 val accuracy: 0.3
lr 2.000000e-07 reg 4.250000e+04 train accuracy: 0.363143 val accuracy: 0.3
lr 2.000000e-07 reg 4.500000e+04 train accuracy: 0.361551 val accuracy: 0.3
lr 2.000000e-07 reg 4.750000e+04 train accuracy: 0.356163 val accuracy: 0.3
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.351714 val accuracy: 0.3
best validation accuracy achieved during cross-validation: 0.405000
```

## 可视化交叉验证结果

```
1. 
2. import math
3. x_scatter = [math.log10(x[0]) for x in results]
4. y_scatter = [math.log10(x[1]) for x in results]
5. 
6. #画出训练准确率
7. marker_size = 100
8. colors = [results[x][0] for x in results]
9. plt.subplot(2, 1, 1)
10. plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
11. plt.colorbar()
12. plt.xlabel('log learning rate')
13. plt.ylabel('log regularization strength')
14. plt.title('CIFAR-10 training accuracy')
15. 
16. #画出验证准确率
17. colors = [results[x][1] for x in results] # default size of markers is
18. plt.subplot(2, 1, 2)
19. plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
20. plt.colorbar()
21. plt.xlabel('log learning rate')
22. plt.ylabel('log regularization strength')
23. plt.title('CIFAR-10 validation accuracy')
24. plt.tight_layout() # 调整子图间距
25. plt.show()
```



## 在测试集上评价最好的svm的表现

```

1. y_test_pred = best_svm.predict(X_test)
2. test_accuracy = np.mean(y_test == y_test_pred)
3. print('linear SVM on raw pixels final test set accuracy: %f' % test_ac

```

输出

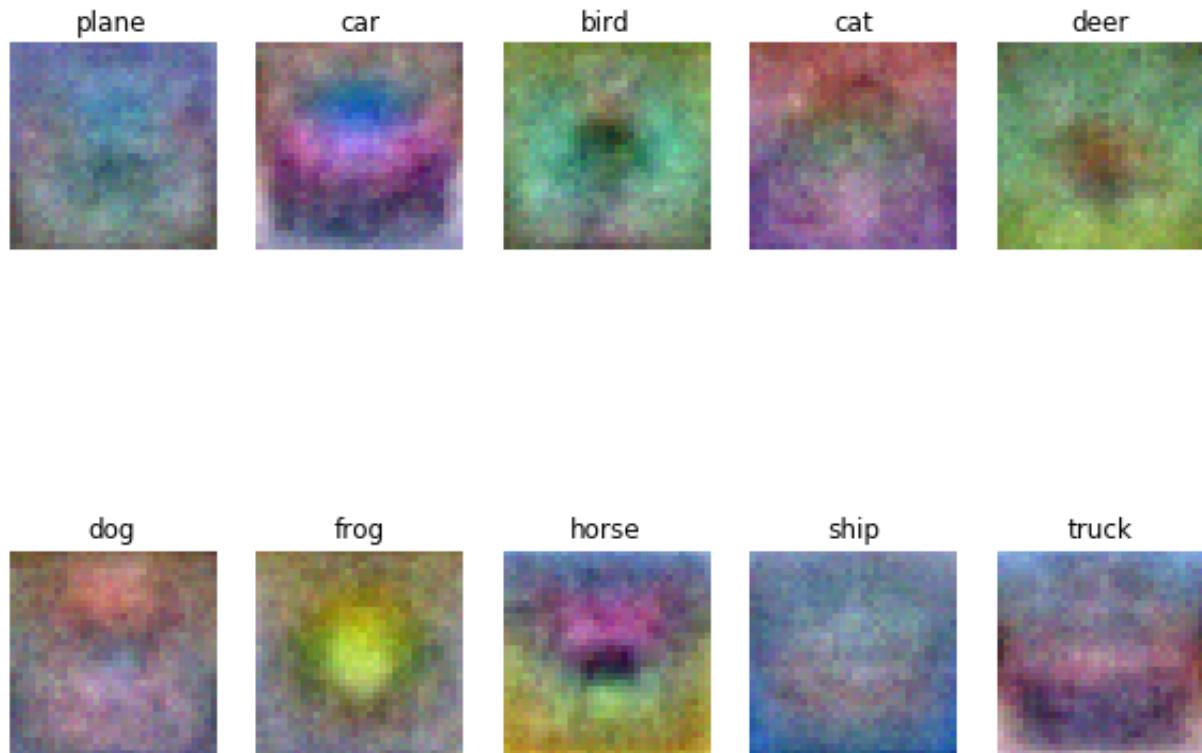
```
linear SVM on raw pixels final test set accuracy: 0.382000
```

结果偏低，不过考虑到是一个10分类问题，均匀分布下，乱猜的结果是0.1，所以还是有那么一点意思的。

```

1. #对于每一类，可视化学习到的权重
2. #依赖于你对学习权重和正则化强度的选择，这些可视化效果或者很明显或者不明显。
3. w = best_svm.W[:-1,:] # strip out the bias
4. w = w.reshape(32, 32, 3, 10)
5. w_min, w_max = np.min(w), np.max(w)
6. classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
7. for i in range(10):
8.     plt.subplot(2, 5, i + 1)
9.
10.    # Rescale the weights to be between 0 and 255
11.    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
12.    plt.imshow(wimg.astype('uint8'))
13.    plt.axis('off')
14.    plt.title(classes[i])

```



## 随堂练习 2:

描述你的SVM可视化图像，给出一个简单的解释

### 答案：

将学习到的权重可视化，从图像可以看出，权重是用于对原图像进行特征提取的工具，与原图像关系很大。很朴素的思想，在分类器权重向量上投影最大的向量得分应该最高，训练样本得到的权重向量，最好的结果就是训练样本提取出来的共性的方向，类似于一种模板或者过滤器。