最近在刷cs231n的课程和作业，在这里分享下自己的学习过程，同时也希望能够得到大家的指点。

写在前面：

1. 这仅仅是自己的学习笔记，如果侵权，还请告知;
2. 代码是参照lightaime的github，在其基础之上做了一些修改;
3. 在我之前，已有前辈在他的知乎上分享过类似内容;
4. 讲义是参照杜客等人对cs231n的中文翻译。

温馨提醒：

1. 这篇文档是建立在你已经知道线性分类器、随机梯度下降法（SGD）的基本原理，如果在阅读本篇文档过程中有些原理不是很清楚，请移步温习下相关知识（参阅cs231n讲义）；
2. 文档的所有程序是使用python3.5实现的，如果你是python2的用户，可能要对代码稍作修改；
3. 文章频繁提到将代码保存到`.py`中，是为了方便接下来的模块导入，希望读者可以理解。

往期文章：
knn分类器

使用线性svm完成cifa10的分类工作，主要有以下几个内容：

1. 模型构建；
2. 数据处理；
3. 梯度计算和检验；
4. 训练和预测；
5. 权重可视化。

1 svm线性分类器构建

对该分类器的构建，主要涉及梯度和损失函数计算、训练和预测模型这两个方面，我们分别来分别对其进行实现。

1.1 损失函数和梯度的计算

损失函数的计算的公式如下（这里是加入**正则项**之后的损失函数值，正则化惩罚可以带来很多良好的性质）：

$$L=\frac{1}{N}\sum_{i}\sum_{j\neq y_i}[max(0,f(x_i;W)_j-f(x_i;W)_{y_i}+\Delta)]+\lambda\sum_{k}\sum_{l}W_{k,l}^2 \tag{1}$$

梯度计算方法如下：

$$\frac{df(x)}{dx}=\lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \tag{2}$$

$$\nabla_{w_{y_i}}L_i=-(\sum_{j\neq y_i}1(w_j^Tx_i-w_{y_i}^Tx_i+\Delta>0))x_i \tag{3}$$

如上所示，梯度更新有两种方法，(2)式数值计算 (3)式微分分析计算，也可看成是矩阵操作。

我们将上述分别用数值方式和矩阵方式进行了实现，代码如下：

```
import numpy as np
from random import shuffle

def svm_loss_naive(W,X,y,reg): #1
    dW=np.zeros(W.shape)
    num_classes=W.shape[1]
    num_train=X.shape[0]
    loss=0.0

    for i in range(num_train):
        scores=X[i].dot(W)
        correct_class_score=scores[y[i]] #1.1
        for j in range(num_classes):
            if j ==y[i]:
                continue
            margin=scores[j]-correct_class_score+1 # 1.2
            if margin> 0:
                loss+=margin
                dW[:,j]+=X[i].T
                dW[:,y[i]]+=-X[i].T
    loss /=num_train
    dW /=num_train
    loss+=0.5*reg*np.sum(W*W)
    dW+=reg*W
    return loss,dW

def svm_loss_vectorized(W,X,y,reg): #2

    num_train = X.shape[0]
    num_classes = W.shape[1]
```

```
    scores = X.dot(W)
    correct_class_scores = scores[range(num_train), list(y)].reshape(-1,
1)
    margins = np.maximum(0, scores - correct_class_scores + 1)
    margins[range(num_train), list(y)] = 0   #2.1
    loss = np.sum(margins) / num_train + 0.5 * reg * np.sum(W * W)

    coeff_mat = np.zeros((num_train, num_classes))
    coeff_mat[margins > 0] = 1
    coeff_mat[range(num_train), list(y)] = 0
    coeff_mat[range(num_train), list(y)] = -np.sum(coeff_mat, axis=1)

    dW = (X.T).dot(coeff_mat)
    dW = dW / num_train + reg * W
    return loss, dW
```

代码解释：#1表示采用数值方式计算损失函数和梯度；#1.1计算正确分类的分数；#1.2中的 `1` 是公式中的 `delta` 值，一般使用1；#2采用矩阵的方式计算损失函数和梯度；#2.1是不计算 `y_i` 。
此外，上面的两个函数输入相同， `W` ，(D，K)维权重，K表示类别数，D表示每个图像维度，如D=32323=3072； `X` ，(N，D)维输入图像，N表示图像数，每个图像都是D*1的列向量； `y` ，(N,)维标签，其中数组的每个数取值为0……k-1，代表第k类； `reg` 表示正则化系数。
我们将这份代码保存到 `svm.py` 中，方便我们后续的调用。
这样我们就完成了使用两种方法计算损失函数和梯度。

1.2 分类器的构建

构建的分类器模型除了损失函数和梯度外，还需要构建训练和预测模型，其中训练模型是采用随机梯度下降法来进行梯度更新的。
代码如下：

```
import numpy as np
from svm import *


class LinearClassifier:
    def __init__(self):
        self.W = None

    def train(self, X, y, learning_rate=1e-3, reg=1e-5,
num_iters=100,batch_size=200, verbose=False):  #1
```

```
10
11          num_train,dim = X.shape
12          num_classes = np.max(y) + 1
13          if self.W is None:
14
15              self.W = 0.001 * np.random.randn(dim, num_classes)
16
17          loss_history = []
18          for it in range(num_iters):
19              X_batch = None
20              y_batch = None
21
22              batch_idx = np.random.choice(num_train,
batch_size,replace=True)
23              X_batch = X[batch_idx]
24              y_batch = y[batch_idx]
25
26              loss, grad = self.loss(X_batch, y_batch, reg)
27
28              loss_history.append(loss)
29
30              self.W += -1 * learning_rate * grad
31
32              if verbose and it % 100 == 0:
33                  print('iteration %d / %d: loss %f' % (it, num_iters,
loss))
34
35          return loss_history
36
37      def predict(self, X):  #2
38
39          y_pred = np.zeros(X.shape[1])
40          scores = X.dot(self.W)
41          y_pred = np.argmax(scores, axis = 1)
42          return y_pred
43
44      def loss(self, X_batch, y_batch, reg):
45          pass
46
47  class LinearSVM(LinearClassifier):  #3
48
49      def loss(self, X_batch, y_batch, reg):
50          return svm_loss_vectorized(self.W, X_batch, y_batch, reg)
51
```

代码解释：#1采用随机梯度下降法（SGD）进行训练，该方法在后面会展开介绍，输入参数 `x`，`y`，`reg` 在前节已经叙述，`learning_rante` 表示学习率，`num_iters` 表示迭代次数，`batch_size` 表示批尺寸，`verbose` 为True时显示中间迭代过程（使用过深度学习一种框架的读者应该对这些参数都不陌生的），输出是每次迭代的损失函数值；#2 预测类别；#3定义一个 `LinearClassifier` 的子类。
我们将其保存到 `linear_classifier.py` 中，方便我们第4章的调用。（**此外，这个函数在下一个作业：softmax分类器中也将被使用**）

2 数据处理

数据转化数组的方式，同knn算法，我们只需要数据加入路径，然后下载下来即可，代码如下：

```
from __future__ import absolute_import
import sys
sys.path.append("..") #1
import random
import numpy as np
from knn.data_utils import load_cifar10

cifar10_dir='..\knn\cifar-10-batches-py'
x_train,y_train,x_test,y_test=load_cifar10(cifar10_dir)
```

```
#验证结果是否正确
print('\n验证结果是否正确')
print('training data shape: ',x_train.shape)
print('training labels shape: ',y_train.shape)
print('test data shape: ',x_test.shape)
print('test labels shape: ',y_test.shape)
```

代码解释：#1的加入是 `data_utils` 和该程序不在同一级目录下，所以导入时需要先返回上一级目录，读者可根据自己的路径进行调整，可参考我的笔记。
我们可以打印输出下，结果如下：

```
training data shape : (50000, 32, 32, 3)
training labels shape : (50000,)
test data shape : (10000, 32, 32, 3)
test labels shape : (10000,)
```

在knn算法中，我们还对cifar10的图片进行了展示，因为内容相同，这里我们就不再进行展示。
我们知道，knn不具有显示的学习过程，svm分类器则不同，它通过训练学习参数W和b，将其保存。训练完成，训练数据就可以丢弃，留下学习到的参数即可。之后一个测试图像可以简单地输入函数，并基于计算出的分类分值来进行分类。而参数的学习过程就是训练过程。
在机器学习中，还有一个必须要重视的问题，那就是过拟合，为了判断是否发生过拟

合，我们从训练集中抽取一部分作为验证集，所以我们的数据集就分为了训练集、验证集和测试集（每个的含义这里我们就不再叙述，相信网上有很多前辈已经解释的很清楚了）代码如下：

```
num_training=49000
num_validation=1000
num_test=1000
num_dev=500

mask=range(num_training,num_training+num_validation)
x_val=x_train[mask]
y_val=y_train[mask]
mask=range(num_training)
x_train=x_train[mask]
y_train=y_train[mask]
mask=np.random.choice(num_training,num_dev,replace=False)
x_dev=x_train[mask]
y_dev=y_train[mask]
mask=range(num_test)
x_test=x_test[mask]
y_test=y_test[mask]
```

```
print('\n验证分离验证集结果是否正确')
print('training data shape: ',x_train.shape)
print('training labels shape: ',y_train.shape)
print('validation data shape: ',x_val.shape)
print('validation data shape: ',y_val.shape)
print('test data shape: ',x_test.shape)
print('test labels shape: ',y_test.shape)
```

代码解释：我们这里处理训练集、验证集、测试集之外又从训练集中随机选择500个样本作为development set，在最终的训练和预测之前，我们都使用这个小的数据集，当然，直接使用完整的训练集也是可以的，不过就是花费的时间有点多。
注意：这里需要先写验证集，再写训练集，否则会报错超出范围！！
我们可以打印输出下，结果如下：

```
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,)
```

我们可以看到，训练集有49000张图片，验证集有1000张图片，测试集有1000张图片。
同knn算法介绍的，我们先将每个图片的三维数组拉成一维向量：

```
x_train=np.reshape(x_train,(x_train.shape[0],-1))
x_val=np.reshape(x_val,(x_val.shape[0],-1))
```

```
x_test=np.reshape(x_test,(x_test.shape[0],-1))
x_dev=np.reshape(x_dev,(x_dev.shape[0],-1))
```

```
print('\n验证三维到一维的转换结果是否正确')
print('training data shape: ',x_train.shape)
print('validation data shape: ',x_val.shape)
print('test data shape: ',x_test.shape)
print('development data shape: ',x_dev.shape)
```

我们输出检查下：

```
training data shape : (49000, 3072)
validation data shape : (1000, 3072)
test data shape : (1000, 3072)
development data shape : (500, 3072)
```

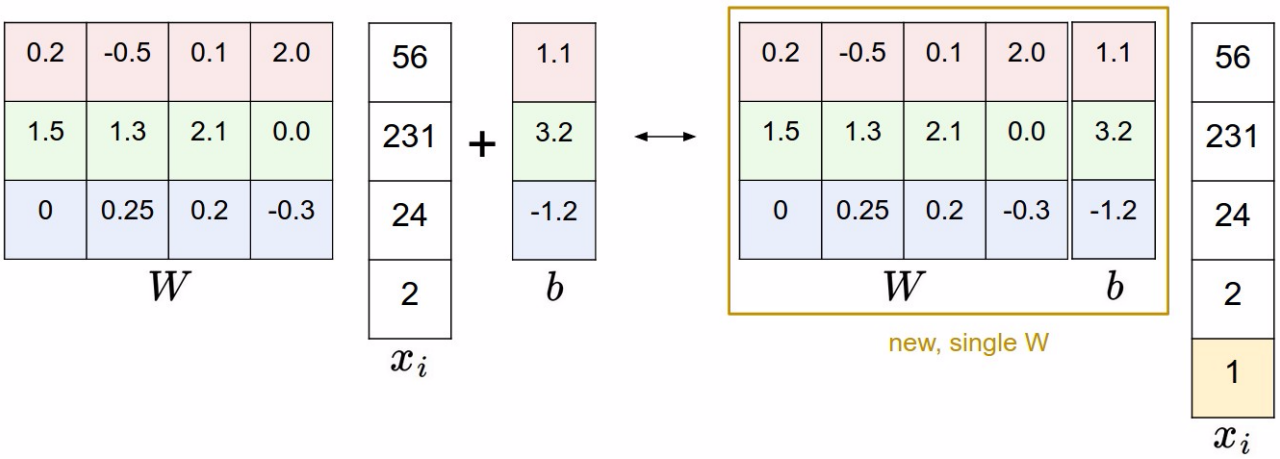
我们模型要求输入的维度也是（N,D）形式的，满足要求，不错，可以奖励下自己一朵小花！

在将我们的数据用于训练和预测之前，我们需要对数据进行归一化处理，这里是对每个特征减去平均值来中心化（讲究的话，还可以再除以方差），代码如下：

```
mean_image=np.mean(x_train,axis=0)
x_train-=mean_image
x_val-=mean_image
x_test-=mean_image
x_dev-=mean_image
```

注意：这里减的均值，是训练集的均值，也就是说训练集、验证集、测试集都需要减去训练集的均值。

等等，还没有完呢！还记得这张图吗，权重矩阵其实是w和b的，因此我们需要对x增加一个维度。




```
x_train=np.hstack([x_train,np.ones((x_train.shape[0],1))])
x_val=np.hstack([x_val,np.ones((x_val.shape[0],1))])
x_test=np.hstack([x_test,np.ones((x_test.shape[0],1))])
```

```
4 x_dev=np.hstack([x_dev,np.ones((x_dev.shape[0],1))])
```

输出检查下：

```
1 (49000, 3073) (1000, 3073) (500, 3073) (1000, 3073)
```

这下我们的完成了数据处理这块，是不是觉得很激动，好的，接下来就可以用已经处理好的数据进行训练和预测了。

```
#输出检查
print('\n验证将b加入到W中的转换结果是否正确')
print('training data shape: ',x_train.shape)
print('validation data shape: ',x_val.shape)
print('test data shape: ',x_test.shape)
print('development data shape: ',x_dev.shape)
```

3 梯度计算和梯度检验

3.1 梯度计算

有了数据，我们就可以使用我们在第1章构建的函数来计算损失函数和梯度了。
我们先来使用数值计算的方式来计算下，代码如下：

```
1 w=np.random.randn(3073,10)*0.0001
2 loss,grad=svm_loss_naive(w,x_dev,y_dev,0.00001)
3 print('loss is : %f '% loss)
```

得到的损失函数值如下：

```
1 loss is : 9.651409
```

因为随机产生参数，每次运行结果会略有变化

3.2 梯度检验

用公式计算梯度速度很快，唯一不好的就是实现的时候容易出错。为了解决这个问题，在实际操作时常常将分析梯度法的结果和数值梯度法的结果作比较，以此来检查其实现的正确性，这个步骤叫做梯度检查。
那么我们计算的梯度是否是正确的呢？这就需要使用梯度检验了，梯度检验公式如下：

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} (use\ instead)$$

其中，h是一个很小的数字，在实践中近似为1e-5。

我们这里使用相对误差来比较数值梯度和解析梯度的差，这里放一张cs231n讲义对这

里的解释。

$$\frac{|f'_a - f'_n|}{max(|f'_a|, |f'_n|)}$$

上式考虑了差值占两个梯度绝对值的比例。注意通常相对误差公式只包含两个式子中的一个（任意一个均可），但是我更倾向取两个式子的最大值或者取两个式子的和。这样做是为了防止在其中一个式子为0时，公式分母为0（这种情况，在ReLU中是经常发生的）。然而，还必须注意两个式子都为零且通过梯度检查的情况。在实践中：

- 相对误差>1e-2：通常就意味着梯度可能出错。
- 1e-2>相对误差>1e-4：要对这个值感到不舒服才行。
- 1e-4>相对误差：这个值的相对误差对于有不可导点的目标函数是OK的。但如果目标函数中没有kink（使用tanh和softmax），那么相对误差值还是太高。
- 1e-7或者更小：好结果，可以高兴一把了。

要知道的是网络的深度越深，相对误差就越高。所以如果你是在对一个10层网络的输入数据做梯度检查，那么1e-2的相对误差值可能就OK了，因为误差一直在累积。相反，如果一个可微函数的相对误差值是1e-2，那么通常说明梯度实现不正确。

我们来看下cs231n提供的梯度检验程序：

```
import random
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-5):

    for i in range(num_checks):
        ix = tuple([randrange(m) for m in x.shape])
                    random.randrange(m)

        oldval = x[ix]
        x[ix] = oldval + h
        fxph = f(x)
        x[ix] = oldval - h
        fxmh = f(x)
        x[ix] = oldval

        grad_numerical = (fxph - fxmh) / (2 * h)
        grad_analytic = analytic_grad[ix]
        rel_error = abs(grad_numerical - grad_analytic) /
(abs(grad_numerical) + abs(grad_analytic))
        print('numerical: %f analytic: %f, relative error: %e' %
(grad_numerical, grad_analytic, rel_error))
```

该程序是为了计算我们前面自己计算的梯度和采用数学方法计算的差别，总共计算十

次。这份程序我们保存在 `gradient_check.py` 文件中。

我们对加入正则项的梯度进行检验：

```
from gradient_check import grad_check_sparse
loss,grad=svm_loss_naive(w,x_dev,y_dev,1e2)
f=lambda w:svm_loss_naive(w,x_dev,y_dev,1e2)[0]
grad_numerical=grad_check_sparse(f,w,grad)
```

得到计算结果如下：

```
numerical: 13.206280 analytic: 13.206280, relative error: 2.098333e-12
numerical: -15.618753 analytic: -15.618753, relative error: 8.239678e-12
numerical: 15.516273 analytic: 15.516273, relative error: 2.278794e-13
numerical: 5.278812 analytic: 5.278812, relative error: 9.194210e-12
numerical: -0.453635 analytic: -0.453635, relative error: 3.158927e-10
numerical: -14.917059 analytic: -14.917059, relative error: 1.076384e-12
numerical: -20.384190 analytic: -20.384190, relative error: 2.866161e-13
numerical: 4.819874 analytic: 4.819874, relative error: 1.054130e-10
numerical: 1.975905 analytic: 1.975905, relative error: 1.700128e-10
numerical: 9.641206 analytic: 9.641206, relative error: 3.015532e-11
numerical: 12.458271 analytic: 12.458271, relative error: 1.153644e-11
numerical: -8.630398 analytic: -8.630398, relative error: 9.508731e-12
numerical: 18.135109 analytic: 18.135109, relative error: 1.818632e-11
numerical: 7.032774 analytic: 7.032774, relative error: 3.613473e-11
numerical: 11.923311 analytic: 11.923311, relative error: 3.160776e-11
numerical: 38.975228 analytic: 38.975228, relative error: 1.080064e-11
numerical: -48.845170 analytic: -48.845170, relative error: 7.069181e-12
numerical: -6.814599 analytic: -6.814599, relative error: 2.159613e-11
numerical: -7.739567 analytic: -7.739567, relative error: 3.680076e-11
numerical: -36.394755 analytic: -36.394755, relative error: 6.801889e-12

```

我们可以看到，误差在e-11数量级，还是很小的。不加入正则项的梯度读者也可以自行检验，只要将 `1e2` 换成 `0.0` 即可。

在 `svm.py` 中，我们采用两种方式实现了svm损失函数和梯度的计算，同knn那篇文章，我们也来比较下这两种方式的区别：

```
import time
tic=time.time()
loss_naive,grad_naive=svm_loss_naive(w,x_dev,y_dev,0.00001)
toc=time.time()
print('naive loss: %e computed in %f s'% (loss_naive,toc-tic))
```

```
5
6  tic=time.time()
7  loss_vectorized,grad_vectorized=svm_loss_vectorized(w,x_dev,y_dev,0.00001
   )
8  toc=time.time()
9  print('vectorized loss: %e computed in %f s'% (loss_vectorized,toc-tic))
10
11 print('difference: %f'% (loss_naive-loss_vectorized))
```

使用%e(科学记数法)更精确

输出结果如下：

```
1  naive loss: 8.127167e+00 computed in 0.125835 s
2  vectorized loss: 8.127167e+00 computed in 0.032619 s
3  difference: -0.000000
```

可见，两种计算方式得到的损失函数值是相同的，而采用向量方法计算时间花费少了很多很多，因此我们接下来将使用 `svm_loss_vectorized` （矩阵）方法计算损失函数和梯度。

既然两种方法计算得出的损失函数值是一样的，那么梯度应该也是一样的，我们也就不需要再对第二种方法进行梯度检验了，不放心的读者也可自行检验下，不过损失函数是一维的，而梯度是二维的，可以使用 `np.linalg.norm` 函数来计算范数，其余同上。

通过这章，我们确定了损失函数和梯度的计算方式。

4 模型训练

4.1 梯度更新

现在我们已经知道采用向量方法计算损失函数和梯度效率最高，并且得到的梯度经过我们的验证误差很小，接下来我们将使用随机梯度下降法（SGD）来进行梯度更新，使得损失函数值最小。
代码如下：

```
1  from linear_classifier import LinearSVM
2  svm=LinearSVM()
3  tic=time.time()
4  loss_hist=svm.train(x_train,y_train,learning_rate=1e-
   7,reg=5e4,num_iters=1500,verbose=True)
5  toc=time.time()
```

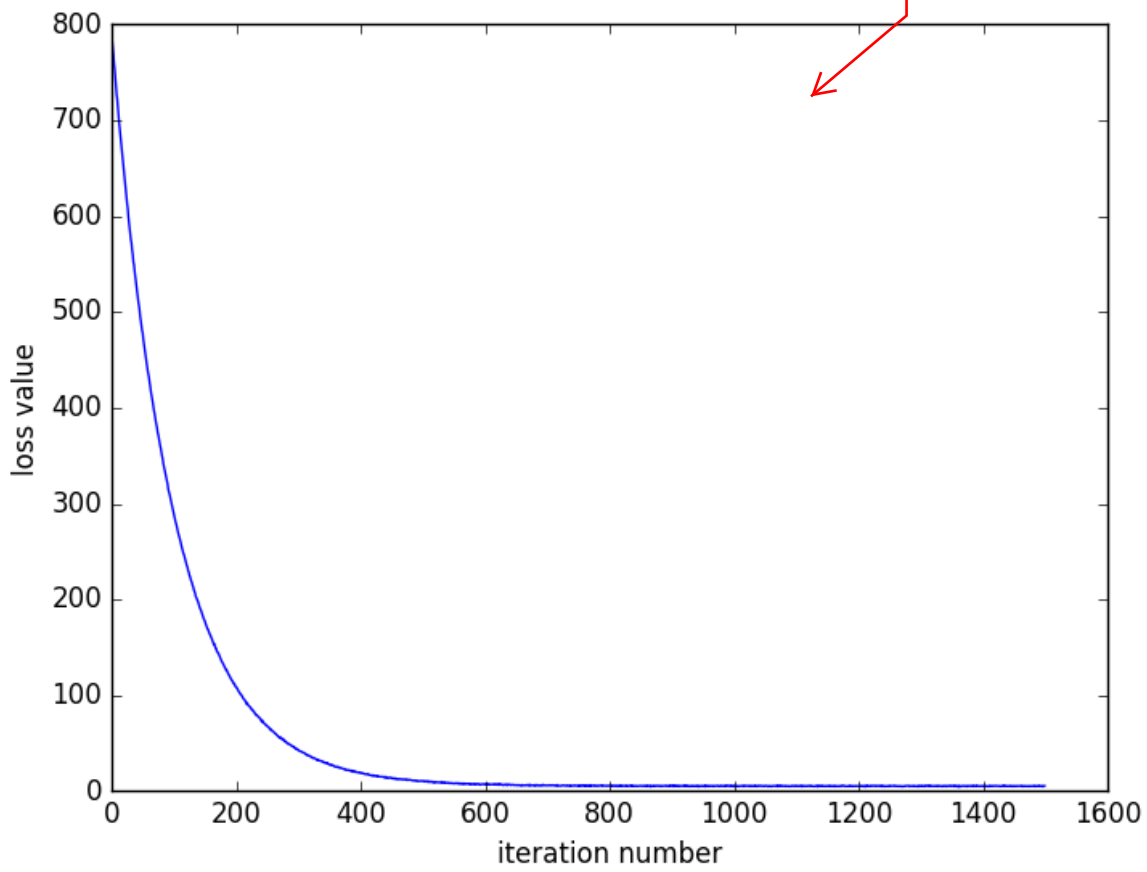
```
print('that took %f s' % (toc-tic))
```

得到的结果如下：

```
iteration 0 / 1500: loss 786.949561
iteration 100 / 1500: loss 286.247383
iteration 200 / 1500: loss 107.371649
iteration 300 / 1500: loss 42.318634
iteration 400 / 1500: loss 18.910315
iteration 500 / 1500: loss 10.107915
iteration 600 / 1500: loss 7.098435
iteration 700 / 1500: loss 5.799740
iteration 800 / 1500: loss 5.791486
iteration 900 / 1500: loss 5.227070
iteration 1000 / 1500: loss 5.235736
iteration 1100 / 1500: loss 5.421294
iteration 1200 / 1500: loss 5.258394
iteration 1300 / 1500: loss 5.222618
iteration 1400 / 1500: loss 5.371492
that took 13.471777 s
```

我们还可以可视化损失函数值，如图：

在linear_classifer.py中添加：
import matplotlib.pyplot as plt
plt.plot(it,loss,'bo-')#可视化损失函数 没有成功



通过该图，我们看到损失函数值在越来越小，已经在发生收敛。

训练完成之后，将参数保存，我们接下来就可以使用这些参数进行预测，并计算准确率，代码如下：

```
y_train_pred=svm.predict(x_train)
print('training accuracy: %f ' % (np.mean(y_train==y_train_pred)))
y_val_pred=svm.predict(x_val)
print('validation accuracy: %f' % (np.mean(y_val==y_val_pred)))
```

得到的结果如下：

```
training accuracy: 0.369755
validation accuracy: 0.382000
```

在训练集上可以达到36%的准确率，验证集可以达到38%的准确率，还是不错的，比我们上一章的knn分类效果好很多了。

4.2 超参数调优

通过手动调整超参数，可以让我们的模型收敛的更快。
学习速率和正则项是超参数，接下来我们就通过交叉验证来选择较好的学习率和正则项系数。
代码如下：

```
learning_rates=[1.4e-7,1.5e-7,1.6e-7]
regularization_strengths=[ (1+i*0.1)*1e4 for i in range(-3,3)] +
[(2+0.1*i)*1e4 for i in range(-3,3)]
results={}
best_val=-1
best_svm=None
for learning in learning_rates:
    for regularization in regularization_strengths:
        svm=LinearSVM()

 svm.train(x_train,y_train,learning_rate=learning,reg=regularization,num_
iters=2000)
        y_train_pred=svm.predict(x_train)
        train_accuracy=np.mean(y_train==y_train_pred)
        print('training accuracy: %f' % (train_accuracy))
        y_val_pred=svm.predict(x_val)
```

```
        val_accuracy=np.mean(y_val==y_val_pred)
        print('validation accuracy: %f' % (val_accuracy))

        if val_accuracy>best_val:
            best_val=val_accuracy
            best_svm=svm
        results[(learning,regularization)]=(train_accuracy,val_accuracy)
for lr , reg in sorted(results):
    train_accuracy,val_accuracy=results[(lr,reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' %
(lr,reg,train_accuracy,val_accuracy))
print('best validation accuracy achieved during cross-validation: %f' %
best_val)
```

代码解释：我们从列举的学习率和正则项中选择验证集正确率最高的超参数，将参数保存到 `best_svm` 中，其中 `results` 存储的是形如 `{(lr,reg):(train_accuracy,val_accuracy)}` 的字典。
得到的运行结果如下：

```
lr 1.400000e-07 reg 7.000000e+03 train accuracy: 0.385020 val accuracy:
0.396000
lr 1.400000e-07 reg 8.000000e+03 train accuracy: 0.390347 val accuracy:
0.386000
lr 1.400000e-07 reg 9.000000e+03 train accuracy: 0.390224 val accuracy:
0.375000
lr 1.400000e-07 reg 1.000000e+04 train accuracy: 0.389776 val accuracy:
0.391000
lr 1.400000e-07 reg 1.100000e+04 train accuracy: 0.391653 val accuracy:
0.387000
lr 1.400000e-07 reg 1.200000e+04 train accuracy: 0.387653 val accuracy:
0.377000
lr 1.400000e-07 reg 1.700000e+04 train accuracy: 0.390735 val accuracy:
0.396000
lr 1.400000e-07 reg 1.800000e+04 train accuracy: 0.377143 val accuracy:
0.391000
lr 1.400000e-07 reg 1.900000e+04 train accuracy: 0.380694 val accuracy:
0.389000
lr 1.400000e-07 reg 2.000000e+04 train accuracy: 0.384143 val accuracy:
0.381000
lr 1.400000e-07 reg 2.100000e+04 train accuracy: 0.386980 val accuracy:
0.389000
lr 1.400000e-07 reg 2.200000e+04 train accuracy: 0.379714 val accuracy:
0.395000
```

```
lr 1.500000e-07 reg 7.000000e+03 train accuracy: 0.392041 val accuracy: 0.395000
lr 1.500000e-07 reg 8.000000e+03 train accuracy: 0.390837 val accuracy: 0.378000
lr 1.500000e-07 reg 9.000000e+03 train accuracy: 0.394204 val accuracy: 0.393000
lr 1.500000e-07 reg 1.000000e+04 train accuracy: 0.386857 val accuracy: 0.386000
lr 1.500000e-07 reg 1.100000e+04 train accuracy: 0.390490 val accuracy: 0.379000
lr 1.500000e-07 reg 1.200000e+04 train accuracy: 0.392490 val accuracy: 0.380000
lr 1.500000e-07 reg 1.700000e+04 train accuracy: 0.383612 val accuracy: 0.386000
lr 1.500000e-07 reg 1.800000e+04 train accuracy: 0.380306 val accuracy: 0.393000
lr 1.500000e-07 reg 1.900000e+04 train accuracy: 0.377388 val accuracy: 0.387000
lr 1.500000e-07 reg 2.000000e+04 train accuracy: 0.386796 val accuracy: 0.395000
lr 1.500000e-07 reg 2.100000e+04 train accuracy: 0.383551 val accuracy: 0.396000
lr 1.500000e-07 reg 2.200000e+04 train accuracy: 0.383592 val accuracy: 0.379000
lr 1.600000e-07 reg 7.000000e+03 train accuracy: 0.389796 val accuracy: 0.390000
lr 1.600000e-07 reg 8.000000e+03 train accuracy: 0.390041 val accuracy: 0.396000
lr 1.600000e-07 reg 9.000000e+03 train accuracy: 0.381286 val accuracy: 0.383000
lr 1.600000e-07 reg 1.000000e+04 train accuracy: 0.391469 val accuracy: 0.392000
lr 1.600000e-07 reg 1.100000e+04 train accuracy: 0.380245 val accuracy: 0.377000
lr 1.600000e-07 reg 1.200000e+04 train accuracy: 0.394143 val accuracy: 0.396000
lr 1.600000e-07 reg 1.700000e+04 train accuracy: 0.375714 val accuracy: 0.387000
lr 1.600000e-07 reg 1.800000e+04 train accuracy: 0.380490 val accuracy: 0.379000
lr 1.600000e-07 reg 1.900000e+04 train accuracy: 0.378714 val accuracy: 0.377000
lr 1.600000e-07 reg 2.000000e+04 train accuracy: 0.382837 val accuracy: 0.384000
lr 1.600000e-07 reg 2.100000e+04 train accuracy: 0.380653 val accuracy: 0.376000
```

```
lr 1.600000e-07 reg 2.200000e+04 train accuracy: 0.371796 val accuracy: 0.394000
best validation accuracy achieved during cross-validation: 0.396000

```

hhhh，一下子打印的有点多，有种凑字数的嫌疑。
如上所示，验证集最高是0.396。

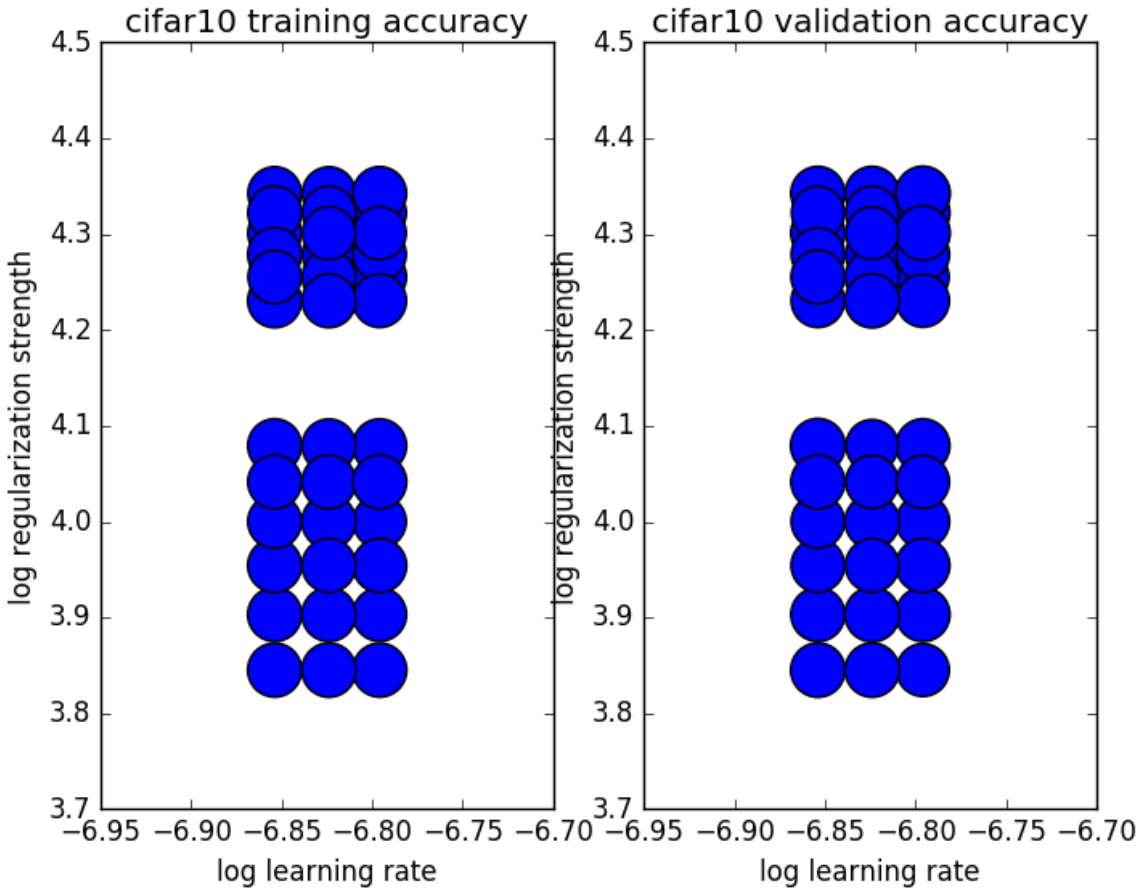
我们还可以以图形的形式将上述结果显示出来，代码如下：

```
import math
import matplotlib.pyplot as plt
x_scatter=[math.log10(x[0]) for x in results] #1
y_scatter=[math.log10(x[1]) for x in results] #2

sz=[results[x][0]*1500 for x in results]  #3
plt.subplot(1,2,1)
plt.scatter(x_scatter,y_scatter,sz)
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('cifar10 training accuracy')

sz=[results[x][1]*1500 for x in results]
plt.subplot(1,2,2)
plt.scatter(x_scatter,y_scatter,sz)
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('cifar10 validation accuracy')
plt.show()
```

代码解释：#1 `x_scatter` 表示学习率；#2 `y_scatter` 表示正则项；#3表示使用面积来代表正确率大小。
得到结果如下图：



如上图，面积的大小代表正确率的大小（哈哈，其实看的不是很明显啊）。

5 模型预测

下面，我们就要使用刚刚保存的最好的模型来进行预测，并输出预测的准确率，代码如下：

```
y_test_pred=best_svm.predict(x_test)
test_accuracy=np.mean(y_test==y_test_pred)
print('linear svm on raw pixels final test set accuracy: %f'%
test_accuracy)
```

输出结果如下：

```
linear svm on raw pixels final test set accuracy: 0.379000
```

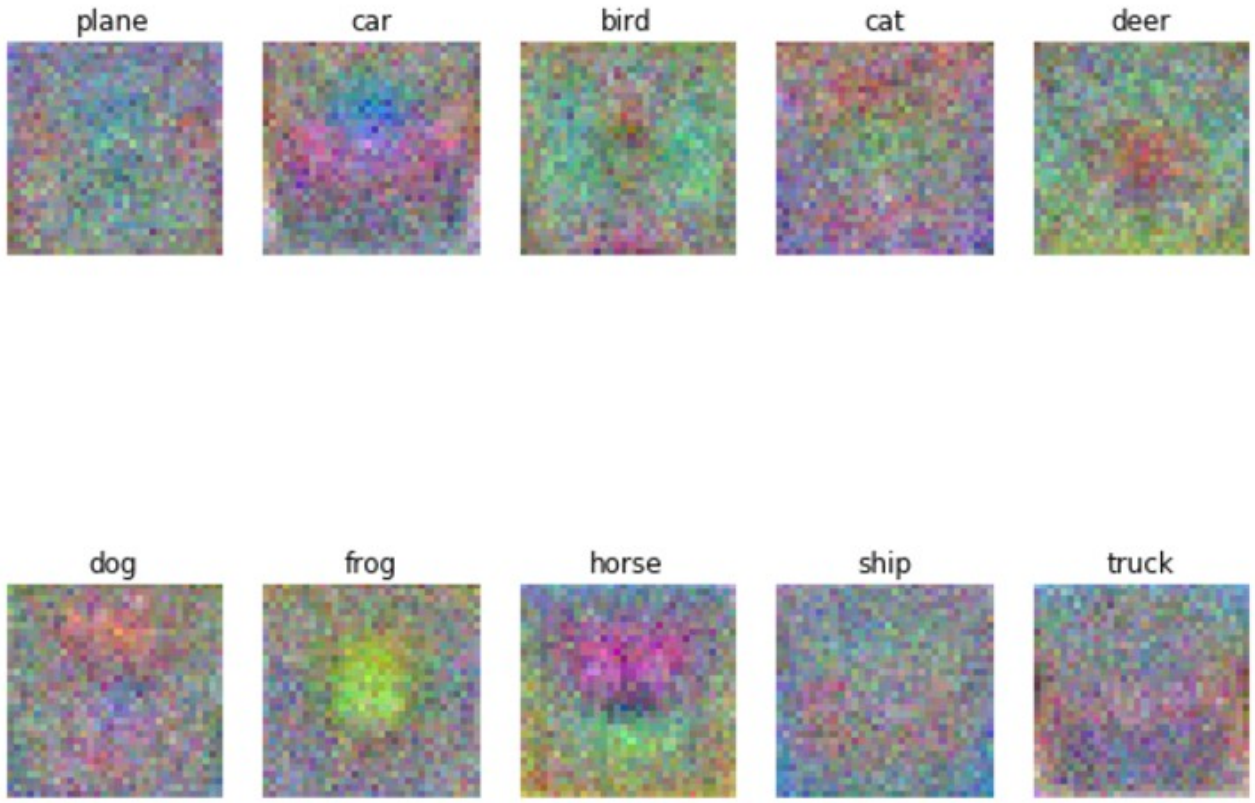
得到了0.379的准确率，还算可以，比我们上一章的knn算法的效果要好很多的。

6 权重可视化

截止到目前，我们差不多已经完成了模型的训练和预测工作，接下来让我们可视化下我们的权重，看看我们的模型到底在学习什么东西。
代码如下：

```
w=best_svm.W[:-1,:]  #1
w=w.reshape(32,32,3, 10)
w_min,w_max=np.min(w),np.max(w)
classes=
['plane','car','bird','cat','deer','dog','frog','horse','ship','truck']
for i in range(10):
    plt.subplot(2,5,i+1)
    wimg=255.0*(w[:,:,:,i].squeeze()-w_min)/(w_max-w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

代码解释：#1是将偏置分离出来，也就是说，我们只可视化权重。
可视化结果如图：



我们来看下权重可视化的结果，比如car这一类，我们可以隐约看到汽车的轮廓，权重参数学习到了这些图像的特征。

7 总结

至此，本篇文章就算是完成了，回想下，我们主要做下以下几个内容：

1. 完成一个使用向量方法计算svm损失函数；
2. 完成一个使用向量方法来分析梯度；
3. 使用数学方法来检查梯度
4. 用验证集来微调学习率和正则项；
5. 使用随机梯度下降法来优化损失函数；
6. 可视化最后学习到的权重