计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：image classification pipeline,		学号：201900130024
日期：2021.10.6	班级： 数据 19	姓名： 刘士渤
Email：liuburger@qq.com		

实验目的：

- understand the basic Image Classification pipeline and the data-driven approach (train/predict stages)
- understand the train/val/test splits and the use of validation data for hyperparameter tuning.
- develop proficiency in writing efficient vectorized code with numpy
- implement and apply a k-Nearest Neighbor (kNN) classifier
- implement and apply a Multiclass Support Vector Machine (SVM) classifier
- implement and apply a Softmax classifier
- implement and apply a Three layer neural network classifier
- understand the differences and tradeoffs between these classifiers
- get a basic understanding of performance improvements from using higher-level representations than raw pixels (e.g. color histograms, Histogram of Gradient (HOG) features)

实验软件和硬件环境：

vscode jupyternotebook

联想拯救者 Y7000p

实验原理和方法：

knn、svm、softmax、three layer net

实验步骤：（不要求罗列完整源代码）

一. knn

1. 完成 k_nearest_neighbor.py

两重循环非常简单，对应位置做点积再开平方就行：

```
dists[i][j]=np.dot((X[i]-self.X_train[j]),(X[i]-self.X_train[j]))**0.5
```

一重循环利用 np.sum 省去了一重循环：

```
dis=np.sqrt(np.sum(np.square(X[i]-self.X_train),axis=1))
dists[i,:]=dis
```

无循环需要另辟蹊径，将欧氏距离展开：

```
XY=np.dot(X,self.X_train.T)
Xi=np.sum(np.square(X),axis=1).reshape([num_test,1])#按行加
Yj=np.sum(np.square(self.X_train),axis=1).reshape([num_train,1])
dists=np.sqrt(Xi-2*XY+np.array(Yj).T)
```

根据参数 k，使用大多数投票的方法确定预测标签

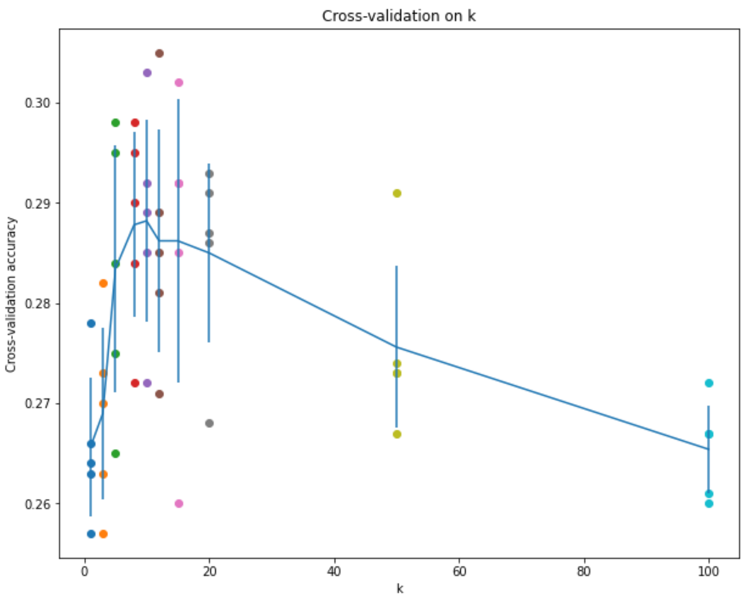
```python
tmp=np.argsort(dists[i])
if k==1:
    y_pred[i]=self.y_train[tmp[0]]
elif k>1:
    for j in range(k):
        closest_y.append(self.y_train[tmp[j]])
    y_num={}
    for label in closest_y:
        if y_num.get(label)==None:
            y_num[label]=1
        else:
            y_num[label]+=1
    y_num=sorted(y_num.items(),key=lambda x:x[1],reverse=True)
    y_pred[i]=y_num[0][0]
```

2. 完成 knn. ipynb

在 k 的集合中用 5 折交叉验证选出最合适的 k：

```python
for k in k_choices:
    k_to_accuracies[k]=[]
    for i in range(0,num_folds):
        classifier.train(tmpx[i],tmpy[i])
        dis=classifier.compute_distances_no_loops(X_train_folds[i])
        y_pred = np.array(classifier.predict_labels(dis, k=k))
        num_correct = np.sum(y_pred == y_train_folds[i])
        accuracy = float(num_correct) / len(y_pred)
        k_to_accuracies[k].append(accuracy)
```

发现在 k=10 的时候效果最好：



用最好的 k 跑，结果达到了 28%

```
Got 144 / 500 correct => accuracy: 0.288000
```

二. svm

1. 完成 linear_classifier

   采用 SGD，所以要有随即成分、梯度下降（-gradient）

```
randi=np.random.choice(num_train,batch_size)
X_batch=X[randi]
y_batch=y[randi]
```

```
self.W-= learning_rate*grad
```

   预测就是选择使目标函数最大的集合

```
y_pred=np.argmax(np.dot(X,self.W),axis=1)
```

2. 完成 linear_svm

   naïve 的关键部分人家已经写好了，自己只需要给 dW 加上正则项的导数：

```
dW+=2*reg*W
```

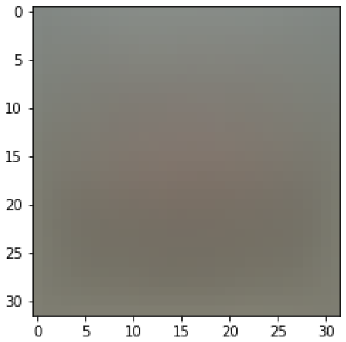
   vectorized 复杂一些：

```
scores=X.dot(W)
#correct_scores的每一行都是正确的分数
correct_scores=scores[np.arange(num_train),y]
correct_scores=np.reshape(np.repeat(correct_scores,num_classes),(num_train,num_classes))
```

```
margin=scores+1.0-correct_scores
#对应相减的地方会是1,则置零
margin[np.arange(num_train),y]=0
#margin现在是loss的初级，要得到loss只需要将正的项加起来
loss=(np.sum(margin[margin>0]))/num_train
loss+=reg*np.sum(W*W)
#分成两类
margin[margin>0]=1
margin[margin<=0]=0
row_sum = np.sum(margin, axis=1)# 1 by N
margin[np.arange(num_train), y] = -row_sum
dW += np.dot(X.T, margin)     # D by C#
#除以N再正则化
dW/=num_train
dW+=2*reg*W
```

3. 完成 svm.ipynb

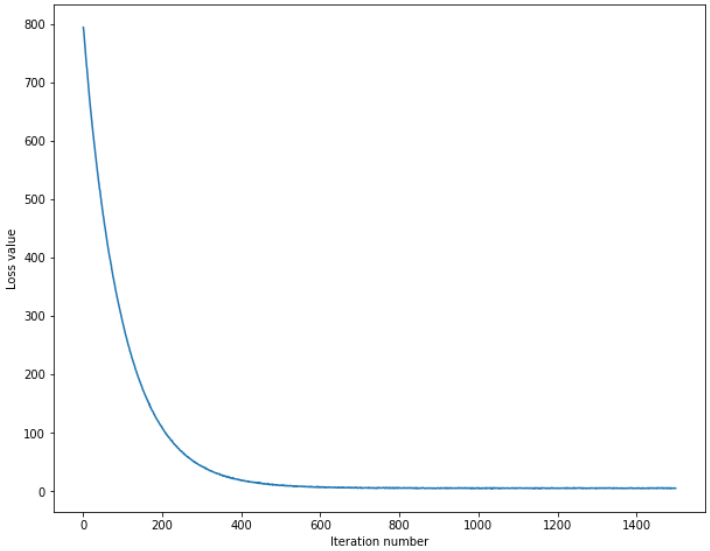
绘制出平均图像：



比对 naïve 和 vectorized 的差异（无差异）：

```
Naive loss: 8.639771e+00 computed in 0.251944s
Vectorized loss: 8.639771e+00 computed in 0.004991s
difference: -0.000000
```

利用 SGD，损失随迭代次数下降：



训练和预测准确率

```
training accuracy: 0.373612
validation accuracy: 0.378000
```

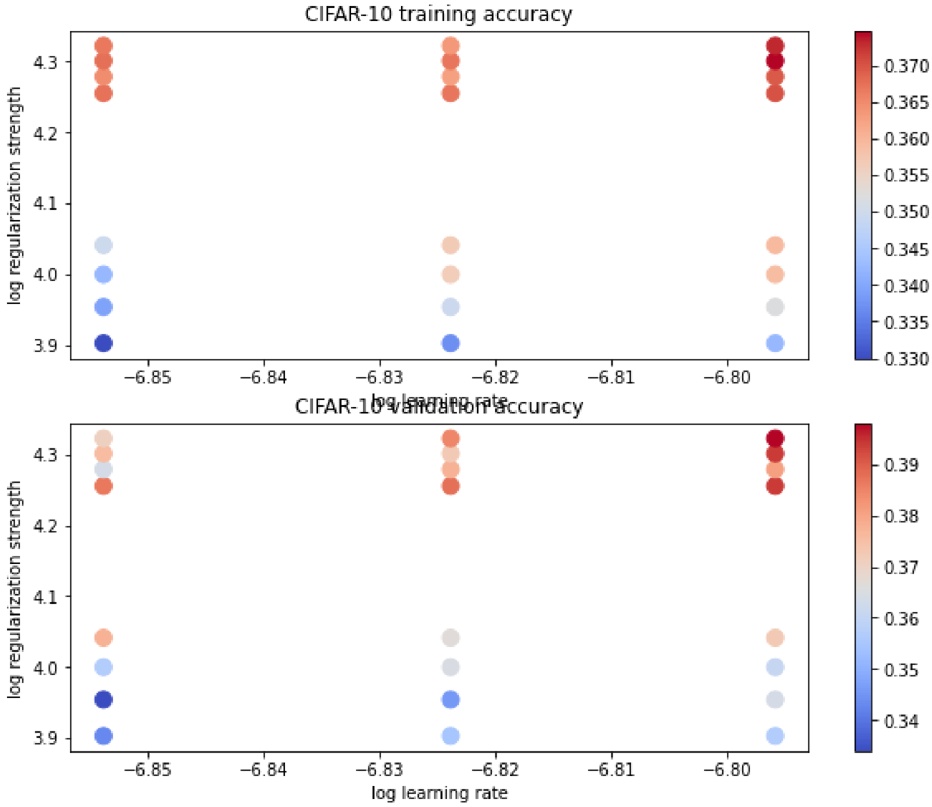
循环调参：

```
for rate in learning_rates:
    for strength in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=rate, reg=strength,
                        num_iters=500, verbose=True)
        y_train_pred = svm.predict(X_train)
        y_val_pred = svm.predict(X_val)
        acc=np.mean(y_val == y_val_pred)
        if acc>best_val:
            best_val=acc
            best_svm=svm
        results[(rate,strength)]=(np.mean(y_train == y_train_pred),acc)
```

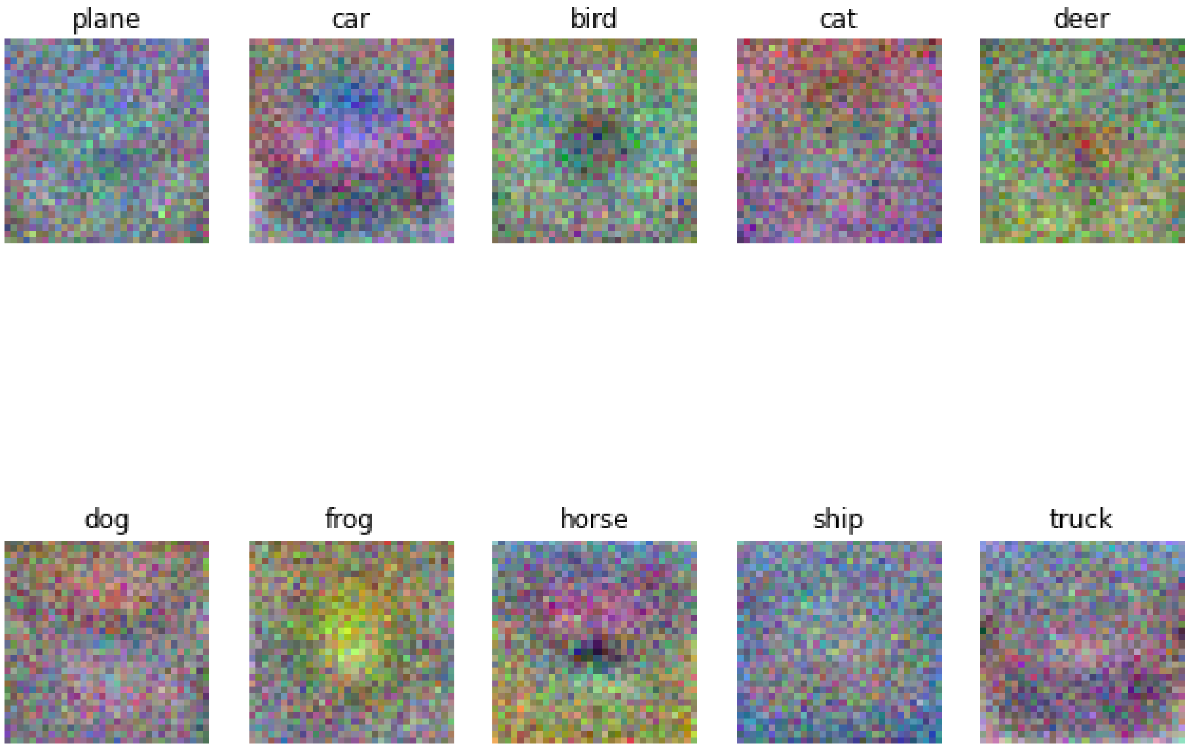
得到最佳的准确率：

```
best validation accuracy achieved during cross-validation: 0.398000
```

在数据集 CIFAR10 上的准确率：



模型可视化（汽车和青蛙还是可以隐约看出来）：



三. softmax

1. 完成 softmax.py

naïve 部分：求 f 函数、损失函数，求导。

```
N,C=X.shape[0],W.shape[1]
for i in range(N):
    f=np.dot(X[i],W)
    f-=np.max(f)#1*C
    s=np.sum(np.exp(f))
    loss+=np.log(s)-f[y[i]]
    dW[:,y[i]]-=X[i]
    for j in range(C):
        dW[:,j]+=np.exp(f[j])/s*X[i]

loss=loss/N+0.5*reg*np.sum(W*W)
dW=dW/N+reg*W
```

vectorized 部分：类似的方法，只是不用循环用矩阵的方法。

```
N=X.shape[0]
f=np.dot(X,W)#N*C
f-=f.max(axis=1).reshape(N,1)
s=np.exp(f).sum(axis=1)
loss=np.log(s).sum()-f[range(N),y].sum()

counts=np.exp(f)/s.reshape(N,1)
counts[range(N),y]-=1
dW=np.dot(X.T,counts)

loss=loss/N+0.5*reg*np.sum(W*W)
dW=dW/N+reg*W
```

2. 完成 softmax.ipynb

比较 naïve 和 vectorized 的 loss 发现无差别：

```
naive loss: 2.427571e+00 computed in 0.142212s
vectorized loss: 2.427571e+00 computed in 0.006065s
Loss difference: 0.000000
Gradient difference: 0.000000
```

循环调参

```
for lr in learning_rates:
    for rs in regularization_strengths:
        softmax=Softmax()
        softmax.train(X_train,y_train,learning_rate=lr,reg=rs,num_iters=500,verbose=True)
        y_pred_train = softmax.predict(X_train)
        acc_train = np.mean(y_pred_train == y_train)
        y_pred_val = softmax.predict(X_val)
        acc_val = np.mean(y_pred_val == y_val)
        results[(lr, rs)] = (acc_train, acc_val)
        if acc_val > best_val:
            best_val = acc_val
            best_softmax = softmax
```

得到最佳的准确率：

```
best validation accuracy achieved during cross-validation: 0.353000
```

模型可视化（效果要比 svm 好得多）：

四. three layer net

1. 完成 neural_net.py

   loss 函数：

   hiddenlayer 就是 y=WX+b 的形式

```
hidden_layer1=np.maximum(0,np.dot(X,W1)+b1)
hidden_layer2=np.maximum(0,np.dot(hidden_layer1,W2)+b2)
scores=np.dot(hidden_layer2,W3)+b3
```

f 函数和 loss 函数（loss 要加正则项）：

```
f=scores-np.max(scores,axis=1,keepdims=True)
loss=-f[range(N),y].sum()+np.log(np.exp(f).sum(axis=1)).sum()
loss=loss/N+0.5*reg*(np.sum(W1*W1)+np.sum(W2*W2)+np.sum(W3*W3))
```

根据 Computational graphs + Backpropagation 原理反向求导：

```
dscore=np.exp(f)/np.exp(f).sum(axis=1,keepdims=True)
dscore[range(N),y]-=1
dscore/=N
#反向传播
grads['W3']=np.dot(hidden_layer2.T,dscore)+reg*W3
grads['b3']=np.sum(dscore,axis=0)
dhidden2=np.dot(dscore,W3.T)
dhidden2[hidden_layer2<=0.00001]=0

grads['W2']=np.dot(hidden_layer1.T,dhidden2)+reg*W2
grads['b2']=np.sum(dhidden2,axis=0)
dhidden1=np.dot(dhidden2,W2.T)
dhidden1[hidden_layer1<=0.00001]=0

grads['W1']=np.dot(X.T,dhidden1)+reg*W1
grads['b1']=np.sum(dhidden1,axis=0)
```

train 函数：

加入随机成分：

```
indices=np.random.choice(num_train,batch_size,replace=True)
X_batch=X[indices]
y_batch=y[indices]
```

参数减去 学习率*梯度：

```
self.params['W1']-=learning_rate*grads['W1']
self.params['b1']-=learning_rate*grads['b1']
self.params['W2']-=learning_rate*grads['W2']
self.params['b2']-=learning_rate*grads['b2']
self.params['W3']-=learning_rate*grads['W3']
self.params['b3']-=learning_rate*grads['b3']
```

predict 函数（取使目标函数 scores 最大的集合）：

```
hidden_layer1=np.maximum(0,np.dot(X,W1)+b1)
hidden_layer2=np.maximum(0,np.dot(hidden_layer1,W2)+b2)
scores=np.dot(hidden_layer2,W3)+b3
y_pred=np.argmax(scores,axis=1)
```

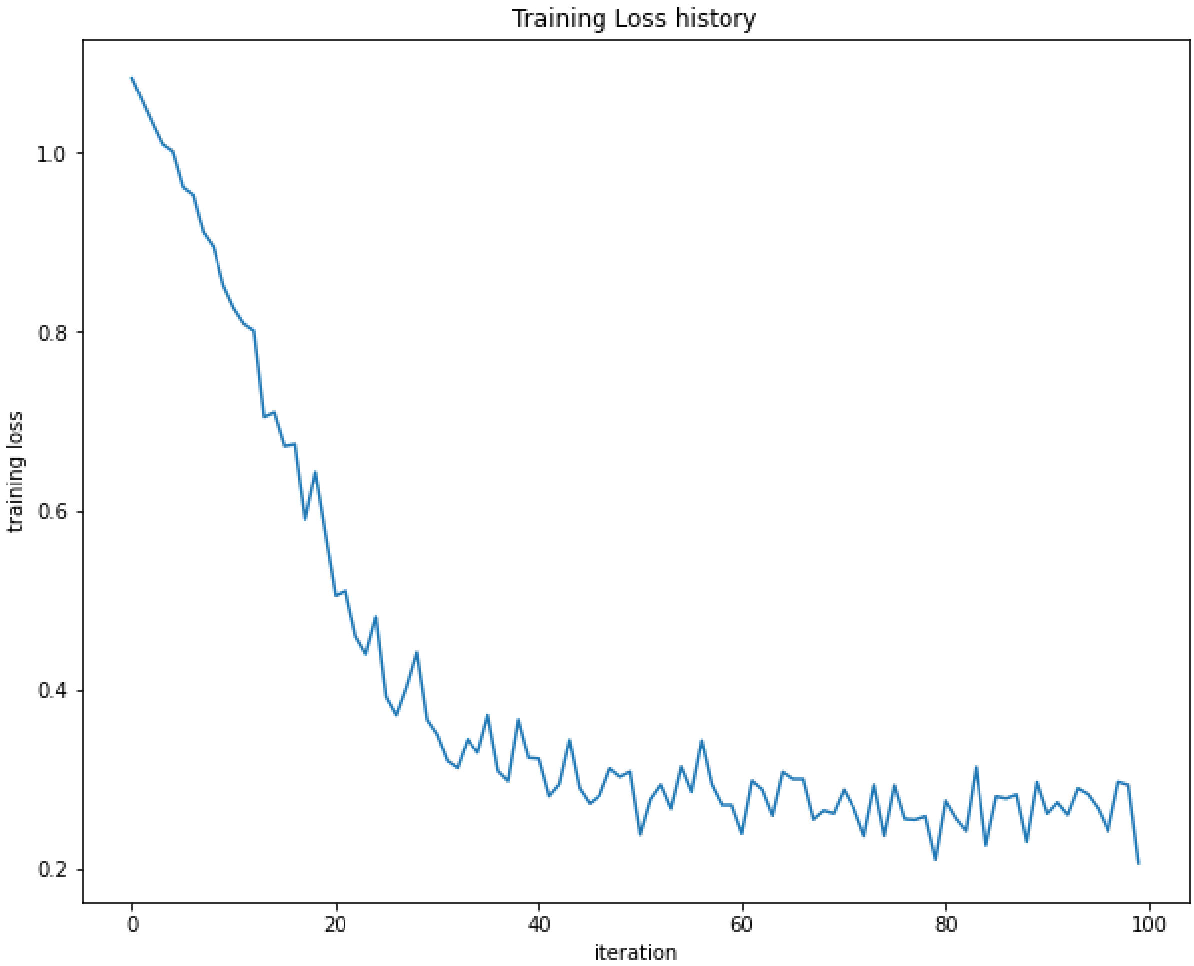
2. 完成 three_layer_net.ipynb

我的 score 和 loss 与正确值相比差异都在预期内：

```
Difference between your scores and correct scores:
4.807962379632267e-08
```

```
Difference between your loss and correct loss:
7.831513215705854e-13
```

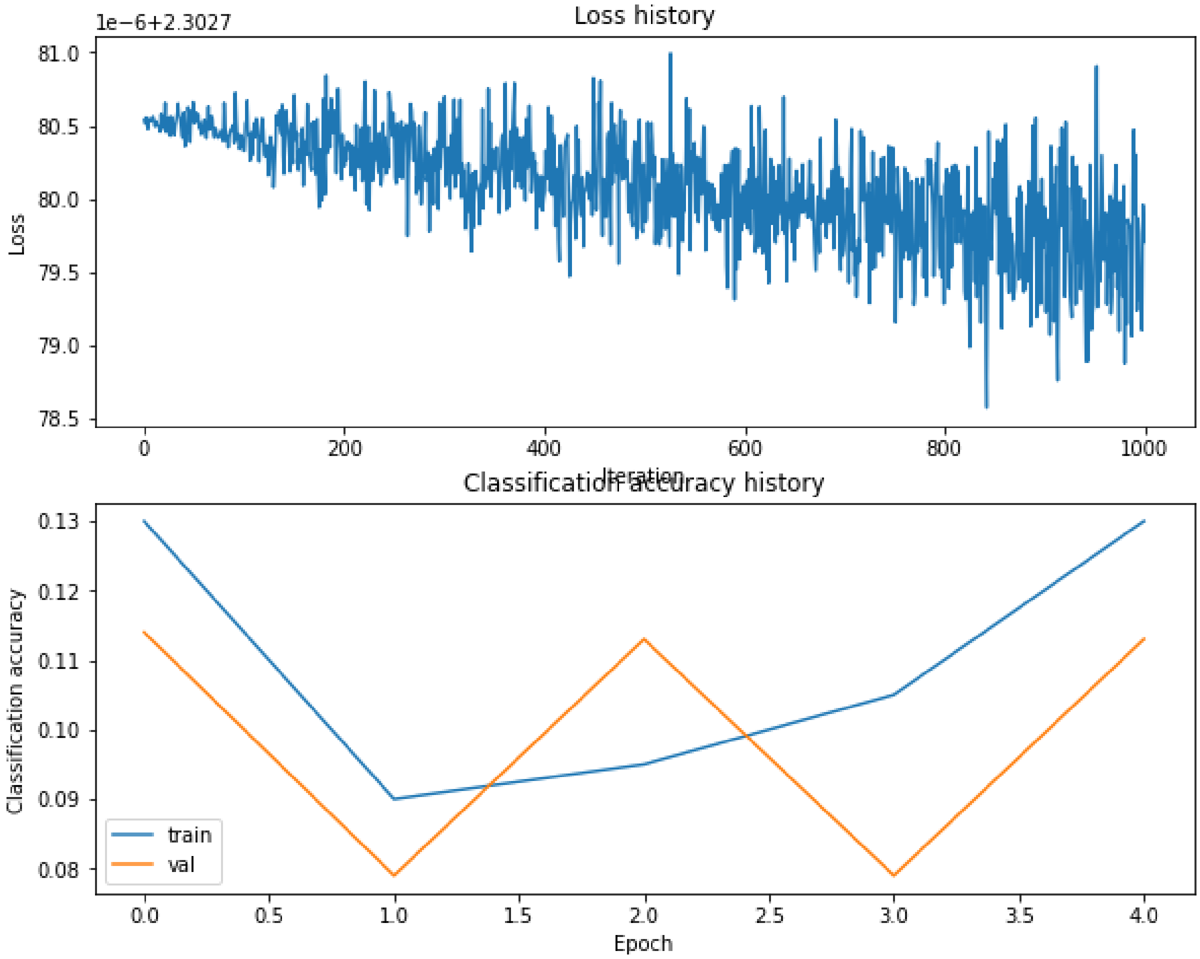
训练误差迭代情况如下：



迭代 1000 次的准确率：

```
iteration 600 / 1000: loss 2.302780
iteration 700 / 1000: loss 2.302780
iteration 800 / 1000: loss 2.302780
iteration 900 / 1000: loss 2.302780
Validation accuracy:  0.113
```

loss 和 classification accuracy 的 history：



调参：

```
learning_rates=[5e-3,7e-3,9e-3]
regularization_strengths=[5e-4,7e-4,9e-3]
for lr in learning_rates:
    for rs in regularization_strengths:
        net=ThreeLayerNet(input_size,hidden_size,num_classes)
        loss_hist=net.train(X_train,y_train,X_val,y_val,
                            num_iters=10000,batch_size=150,
                            learning_rate=lr,learning_rate_decay=0.95,
                            reg=rs,verbose=False)

        val_acc = (net.predict(X_val) == y_val).mean()
        result[(lr,rs)]=val_acc
        if val_acc>best_val:
            best_val=val_acc
            best_net=net
```

模型可视化：

最终准确率及对应的 lr（0.005）和 reg（0.0007）：

```
Test accuracy:  0.528
0.005
0.0007
```

五. features

1. 完成 features.ipynb

on SVM：

调参：

```
for lr in learning_rates:
    for rs in regularization_strengths:
        svm=LinearSVM()
        loss_hist=svm.train(X_train_feats,y_train,learning_rate=lr,reg=rs,
                                        num_iters=1500,verbose=False)
        y_train_pred=svm.predict(X_train_feats)
        train_acc=np.mean(y_train==y_train_pred)
        y_val_pred=svm.predict(X_val_feats)
        val_acc=np.mean(y_val==y_val_pred)
        results[(lr,rs)]=(train_acc,val_acc)
        if(val_acc>best_val):
            best_val=val_acc
            best_svm=svm
```

准确率：

```
✓ 0.1s

0.425
```

on neural_net:

调参（由于运行时间太长，找到最优组合后就注释了列表）：

```python
#learning_rates = [1e-2,0.1,1]
#regularization_strengths = [1e-5,1e-4,1e-3]
learning_rates = [1]
regularization_strengths = [1e-5]
for lr in learning_rates:
    for rs in regularization_strengths:
        net = ThreeLayerNet(input_dim, hidden_dim, num_classes)
        lost_hist = net.train(X_train_feats, y_train, X_val_feats, y_val,
                              num_iters=10000, batch_size=150,
                              learning_rate=lr, reg= rs, learning_rate_decay=0.95,
                              verbose=False)
        val_acc = np.mean(net.predict(X_val_feats) == y_val)
        if val_acc > best_val:
            best_val = val_acc
            best_net = net
        results[(lr,rs)] = val_acc
```

最佳组合（lr，reg）及准确率：

```
(1, 1e-05) 0.565
```

结论分析与体会：

1. 对比 svm 与 neural network 的参数我发现，在大多数情况下：
   svm 的 lr（通常在 1e-7 左右）< neural_net 的 lr（通常在 5e-3 及以上）、
   svm 的 reg（通常在 1e4 左右）> neural_net 的 reg（通常在 1e-4 及以下）、
   svm 的迭代次数（通常 1500 及以下）< neural_net 的迭代次数（动辄上万），
   综上，svm 似乎更好用一些（调一次 neural_net 的参数太烦人了）。

就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1—3 道问答题：

1. 在 features.ipynb 那里，我试了很多 learning_rate 和 regularization_strength，准确率始终只有 0.1 左右（这时候迭代次数是 1500，batch_size 是 200），这让我一度怀疑自己的 neural_net.py 写错了。但是我在之前运行 three_layer_net.ipynb 的准确率达到了 0.52，它同样依托 neural_net.py，证明我的 neural_net.py 是没有问题的。

```python
test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
10]                                                              Python
... 0.103
```

最终我将迭代次数设为 10000，batch_size 设为 150，跑出了 0.56 的准确率，最佳组合是 lr=1，reg=1e-5。

```
test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

[12] ✓ 0.1s

```
... 0.573
```

虽然取得了不错的效果，但是我总觉得舍近求远了，因为用两层的神经网络迭代 1500 次就能跑出 0.57，为什么还要用三层迭代 10000 次甚至更多呢？

```
test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

✓ 0.9s

```
0.561
```