# COMP540 STATISTICAL MACHINE LEARNING

Chuanwenjie Wei, Yiqing Lu

February 20, 2017

## 1 MAP and MLE parameter estimation

- Since $x^{(i)} \sim Ber(\theta)$, we have

$$L(\theta) = \prod_{i=1}^{m} \theta^{x^{(i)}} (1-\theta)^{1-x^{(i)}}$$

By the method of maximum likelihood estimation, we konw that

$$\theta_{MLE} = argmax_{\theta} L(\theta) = argmax_{\theta} LL(\theta)$$

where $LL(\theta)$ denotes the log likelihood of the data set.

So we can derive $\theta_{MLE}$ from

$$\frac{\partial LL(\theta)}{\partial\theta} = \frac{\partial \left[\sum_{i=1}^{m} \left(x^{(i)} log\theta + \left(1-x^{(i)}\right) log(1-\theta)\right)\right]}{\partial\theta} = 0$$

that is,

$$\frac{1}{\theta}\sum_{i=1}^{m} x^{(i)} - \frac{1}{1-\theta}\sum_{i=1}^{m}\left(1-x^{(i)}\right) = 0$$

So

$$\theta_{MLE} = \frac{\sum_{i=1}^{m} x^{(i)}}{m}$$

- Assume $\theta \sim Beta(a,b)$, So

$$p(X \mid \theta) = \pi(\theta) L(\theta) = Beta(a,b) \prod_{i=1}^{m} Ber\left(x^{(i)}, \theta\right)$$

$$= \theta^{a-1}(1-\theta)^{b-1} \prod_{i=1}^{m} \theta^{x^{(i)}} (1-\theta)^{1-x^{(i)}}$$

We know that

$$\theta_{MAP} = argmax_{\theta} P\left(X \mid \theta\right) = argmax_{\theta} logP\left(X \mid \theta\right)$$

Since

$$logP\left(X \mid \theta\right) = log\theta\left(a-1+\sum_{i=1}^{m} x^{(i)}\right) + log\left(1-\theta\right)\left(b-1+\sum_{i=1}^{m}\left(1-x^{(i)}\right)\right)$$

So

$$\frac{\partial logP\left(X \mid \theta\right)}{\partial\theta} = \frac{1}{\theta}\left(a-1+\sum_{i=1}^{m} x^{(i)}\right) - \frac{1}{1-\theta}\left(b-1+\sum_{i=1}^{m}\left(1-x^{(i)}\right)\right) = 0$$

$$\Longrightarrow \theta_{MAP} = \frac{a-1+\sum_{i=1}^{m} x^{(i)}}{a+b-2+m}$$

When $a = b = 1$,

$$\theta_{MAP} = \frac{\sum_{i=1}^{m} x^{(i)}}{m} = \theta_{MLE}$$

## 2 Logistic regression and Guassian Naive Bayes

- For Logistic regression, we have

$$p\left(y=1 \mid x^{(i)}\right) = \frac{1}{1+e^{-\theta^T x^{(i)}}}$$

$$p\left(y=0 \mid x\right) = 1 - \frac{1}{1+e^{-\theta^T x^{(i)}}}$$

- According to Bayes rule,

$$p\left(y=1 \mid x^{(i)}\right) = \frac{p\left(x^{(i)} \mid y=1\right)p\left(y=1\right)}{p\left(x^{(i)}\right)} = \frac{p\left(y=1\right)\prod_{j=1}^{d} p\left(x_j^{(i)} \mid y=1\right)}{\prod_{j=1}^{d} p\left(x_j^{(i)}\right)}$$

Since

$$y \sim Ber\left(\gamma\right),\ x_j/y=1 \sim N\left(\mu_j^1, \sigma_j^2\right),\ x_j/y=0 \sim N\left(\mu_j^0, \sigma_j^2\right)$$

So

$$p\left(y=1\right) = \gamma,$$

$$p\left(x_j^{(i)} \mid y=1\right) = \frac{1}{\sqrt{2\pi}\sigma_j} exp\left\{-\frac{\left(x_j^{(i)}-\mu_j^1\right)^2}{2\sigma_j^2}\right\}$$

$$p\left(x^{(i)}\right) = \gamma\prod_{j=1}^{d} N\left(x_j^{(i)}, \mu_j^1, \sigma_j^2\right) + \left(1-\gamma\right)\prod_{j=1}^{d} N\left(x_j^{(i)}, \mu_j^0, \sigma_j^2\right)$$

Therefore,

$$p\left(y=1 \mid x^{(i)}\right) = \frac{\gamma \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}\sigma_j} exp\left\{-\frac{\left(x_j^{(i)}-\mu_j^1\right)^2}{2\sigma_j^2}\right\}}{\gamma \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}\sigma_j} exp\left\{-\frac{\left(x_j^{(i)}-\mu_j^1\right)^2}{2\sigma_j^2}\right\} + (1-\gamma) \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi}\sigma_j} exp\left\{-\frac{\left(x_j^{(i)}-\mu_j^0\right)^2}{2\sigma_j^2}\right\}}$$

- Assume that class 1 and class 0 are equally likely, that is, $\gamma = \frac{1}{2}$

Then, we have

$$p\left(y=1 \mid x^{(i)}\right) = \frac{1}{1+\prod_{j=1}^{d} exp\left\{-\frac{\left(x_j^{(i)}-\mu_j^0\right)^2-\left(x_j^{(i)}-\mu_j^1\right)^2}{2\sigma_j^2}\right\}}$$

When $\mu_j^0 = -\mu_j^1$, we can rewrite the posterior probability as:

$$\begin{aligned} p\left(y=1 \mid x^{(i)}\right) &= \frac{1}{1+\prod_{j=1}^{d} exp\left\{-\frac{4x_j^{(i)}\mu_j^1}{2\sigma_j^2}\right\}} \\ &= \frac{1}{1+exp\left\{-\sum_{j=1}^{d} \frac{2\mu_j^1}{\sigma_j^2} x_j^{(i)}\right\}} \\ &= \frac{1}{1+e^{-\theta^T x^{(i)}}} \end{aligned}$$

where $\theta = \left(\frac{2\mu_1^1}{\sigma_1^2}, \frac{2\mu_2^1}{\sigma_2^2}, \ldots, \frac{2\mu_d^1}{\sigma_d^2}\right)^T$

So with appropriate parameterization as above, $P(y=1 \mid x)$ for Gaussian Naive Bayes with uniform priors is equivalent to $P(y=1 \mid x)$ for logistic regression.

## 3 Reject option in classifiers

- When$j = 1, \ldots, c$, the average loss of choosing class $j$ is:

$$\begin{aligned} L(\alpha_j \mid x) &= \sum_{k=1}^{c} L(\alpha_j \mid y=k)\, p(y=k \mid x) \\ &= \sum_{k=1, k\neq j}^{c} \lambda_s p(y=k \mid x) + 0 * p(y=j \mid x) \\ &= \lambda_s (1 - p(y=j \mid x)) \end{aligned}$$

When $j = c+1$,

$$L(\alpha_{j+1} \mid x) = \lambda_r$$

Since when the cost for rejects is less than the cost of falsely classifying the object, it might be the optimal action. Then if we decide $y = j$, we not only need $p(y = j \mid x) \geqslant p(y = k \mid x)$ for all $k$, but also $L(\alpha_j \mid x) \leqslant L(\alpha_{j+1} \mid x)$ which means:

$$\lambda_s (1 - p(y = j \mid x)) \quad \leqslant \quad \lambda_r$$

$$\Rightarrow \quad p(y = j \mid x) \geqslant 1 - \frac{\lambda_r}{\lambda_s}$$

• When $\lambda_r/\lambda_s$ approches to 0, the cost of falsely classifying is extremely large, in other words, the cost of rejects is small, so we tend to choose rejects.

However, as $\lambda_r/\lambda_s$ increases, the relative cost of rejection increases and the probability of rejecting decreases. And when $\lambda_r \geqslant \lambda_s$, we would never reject.

## 4 One vs all logistic regression and softmax regression

### 4.1A Implementing a one-vs-all classier for the CIFAR-10 dataset

The code of train function of OVA is in Fig.1.

```
##########################################################################
# Compute the K logistic regression classifiers                          #
# TODO: 7-9 lines of code expected                                       #
##########################################################################
for i in range(len(self.labels)):
    y_new1=np.zeros(len(y))+i
    y_new=(y==y_new1)
     for j in range(len(y)):
         if y[j]==i:
             y_new[j]=1
    sk_logreg = linear_model.LogisticRegression(C=1.0/reg,solver='lbfgs',fit_intercept=False)
    sk_logreg.fit(X,y_new)
    if i==0:
        theta_1=sk_logreg.coef_[0]
    else:
        theta_1=np.vstack([theta_1,sk_logreg.coef_[0]])
theta_opt=theta_1.T
##########################################################################
#                            END OF YOUR CODE                            #
##########################################################################
```

Fig.1 The train function

### 4.1B Predicting with a one-vs-all classier

The code of predict function of OVA is in Fig.2.

```
###########################################################################
# Compute the predicted outputs for X                                     #
# TODO: 1-2 lines of code expected                                        #
###########################################################################
y_pred1=np.dot(X,self.theta)
for i in range(X.shape[0]):
    y_pred[i]=np.argmax(y_pred1[i,:])
###########################################################################
#                           END OF YOUR CODE                              #
###########################################################################
```

Fig.2 The predict function

One_vs_all on raw pixels final test set accuracy: 0.361400. The result of the confution matrix is

```
[[464  58  22  24  19  35  26  60 202  90]
 [ 69 464  18  35  23  31  44  51  91 174]
 [121  64 193  77  96  89 151  89  73  47]
 [ 66  86  78 161  48 193 171  51  63  83]
 [ 65  38 103  64 234  90 194 129  35  48]
 [ 49  64  81 127  81 273 114  89  67  55]
 [ 31  53  67 102  87  78 456  51  29  46]
 [ 53  62  50  46  68  85  66 406  48 116]
 [146  79   8  25   9  34  22  19 543 115]
 [ 59 208  14  22  23  29  59  56 110 420]]
```

When label=0:

| | $y = 0$ | otherwise |
|---|---|---|
| $\hat{y} = 0$ | 464 | 536 |
| otherwise | 659 | 3150 |

$$\Longrightarrow \begin{cases} specificity = 0.855 \\ accuracy = 0.752 \\ sensitivity = 0.413 \end{cases}$$

When label=1:

| | $y = 1$ | otherwise |
|---|---|---|
| $\hat{y} = 1$ | 464 | 536 |
| otherwise | 712 | 3150 |

$$\Longrightarrow \begin{cases} specificity = 0.855 \\ accuracy = 0.743 \\ sensitivity = 0.395 \end{cases}$$

When label=2:

| | $y = 2$ | otherwise |
|---|---|---|
| $\hat{y} = 2$ | 193 | 807 |
| otherwise | 441 | 3421 |

$$\Longrightarrow \begin{cases} specificity = 0.809 \\ accuracy = 0.743 \\ sensitivity = 0.304 \end{cases}$$

When label=3:

| | $y=3$ | otherwise |
|---|---|---|
| $\hat{y}=3$ | 161 | 839 |
| otherwise | 522 | 3453 |

$$\Longrightarrow \begin{cases} specificity = 0.805 \\ accuracy = 0.726 \\ sensitivity = 0.236 \end{cases}$$

When label=4:

| | $y=4$ | otherwise |
|---|---|---|
| $\hat{y}=4$ | 234 | 766 |
| otherwise | 454 | 3380 |

$$\Longrightarrow \begin{cases} specificity = 0.815 \\ accuracy = 0.724 \\ sensitivity = 0.340 \end{cases}$$

When label=5:

| | $y=5$ | otherwise |
|---|---|---|
| $\hat{y}=5$ | 273 | 727 |
| otherwise | 664 | 3341 |

$$\Longrightarrow \begin{cases} specificity = 0.821 \\ accuracy = 0.722 \\ sensitivity = 0.291 \end{cases}$$

When label=6:

| | $y=6$ | otherwise |
|---|---|---|
| $\hat{y}=6$ | 456 | 544 |
| otherwise | 847 | 3158 |

$$\Longrightarrow \begin{cases} specificity = 0.853 \\ accuracy = 0.722 \\ sensitivity = 0.350 \end{cases}$$

When label=7:

| | $y=7$ | otherwise |
|---|---|---|
| $\hat{y}=7$ | 406 | 594 |
| otherwise | 595 | 3208 |

$$\Longrightarrow \begin{cases} specificity = 0.844 \\ accuracy = 0.752 \\ sensitivity = 0.406 \end{cases}$$

When label=8:

| | $y=8$ | otherwise |
|---|---|---|
| $\hat{y}=8$ | 543 | 457 |
| otherwise | 718 | 3071 |

$$\Longrightarrow \begin{cases} specificity = 0.870 \\ accuracy = 0.755 \\ sensitivity = 0.431 \end{cases}$$

When label=9:

| | $y=9$ | otherwise |
|---|---|---|
| $\hat{y}=9$ | 420 | 580 |
| otherwise | 774 | 3194 |

$$\Longrightarrow \begin{cases} specificity = 0.846 \\ accuracy = 0.727 \\ sensitivity = 0.352 \end{cases}$$

So the table comparing the per genre classication performance of each classier is in Table.1.

| | y=0 | y=1 | y=2 | y=3 | y=4 | y=5 | y=6 | y=7 | y=8 | y=9 |
|---|---|---|---|---|---|---|---|---|---|---|
| specificity | 0.855 | 0.855 | 0.809 | 0.805 | 0.815 | 0.821 | 0.853 | 0.844 | 0.870 | 0.846 |
| accuracy | 0.752 | 0.743 | 0.743 | 0.726 | 0.724 | 0.722 | 0.722 | 0.752 | 0.755 | 0.727 |
| sensitivity | 0.413 | 0.395 | 0.304 | 0.236 | 0.340 | 0.291 | 0.350 | 0.406 | 0.431 | 0.352 |

Table.1 Per genre classication performance of each classier

The results are similar because for the whole data set, the amount of every label is almost the same, and the model of every model is the same, so the results of specificity, accuracy and sensitivity are similar.

• **Visualizing the learned parameter matrix** The result of the learned one-vs-all classifier is in Fig.3.

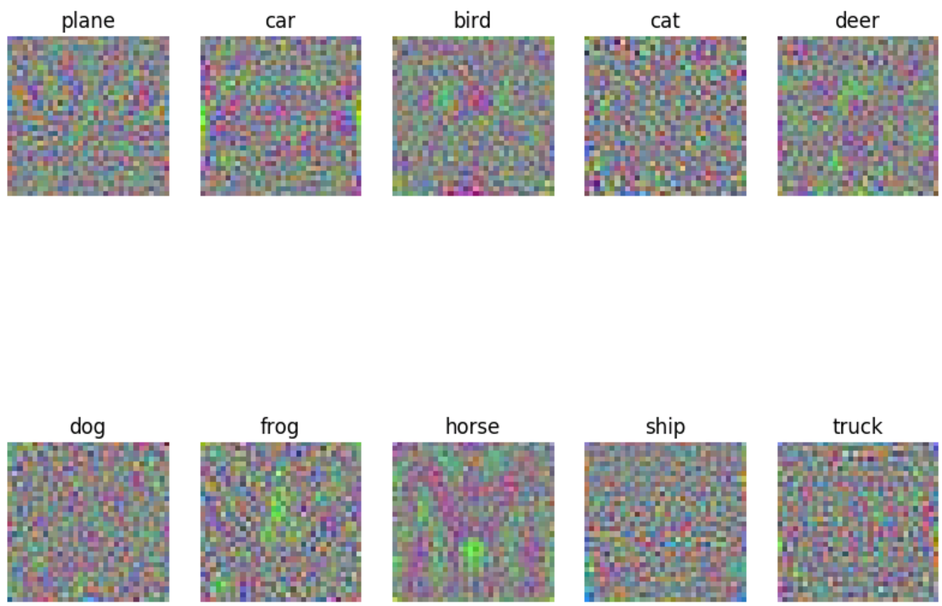


Fig.3 Visualizing the learned one-vs-all classifier

### 4.2 Implementing the loss function for softmax regression (naive version)

The code of the loss function(naive version) for softmax regression is in Fig.4.

```
K=grad.shape[1]
for i in xrange(m):
  X_i =  X[i,:]
  score_i = np.dot(X_i,theta)
  s_max = -score_i.max()
  exp_score = np.exp(score_i+s_max)
  total_score = np.sum(exp_score , axis = 0)
  numerator = np.exp(score_i[y[i]]+s_max)
  denom = np.sum(np.exp(score_i+s_max),axis = 0)
  J = J -np.log(numerator / float(denom))

J = J / float(m) + 0.5 * reg * np.sum(theta**2) / float(m)
```

Fig.4 The loss function(naive version)

The result of loss is: 2.3696905683 . The result should be close to 2.38$=-log_e(0.1)$, because for a large amount of data, the possibility $p(y^{(i)}=k)$ is equal, and because $\theta$ is a matrix of random numbers, so $\frac{exp(\theta^{(k)^T}x^{(i)})}{\sum_{j=1}^{k} exp(\theta^{(j)^T}x^{(i)})} \approx 0.1$. And because $\lambda = 0$ in this case:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} I(y^{(i)}=k)log\frac{exp(\theta^{(k)^T}x^{(i)})}{\sum_{j=1}^{k} exp(\theta^{(j)^T}x^{(i)})} + \frac{\lambda}{m}\sum_{j=0}^{d}\sum_{k=1}^{K}\theta_j^{(k)^2}$$

$$\approx -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} I(y^{(i)}=k)log0.1$$

$$= -\frac{1}{m} * m\sum_{k=1}^{K} I(y^{(i)}=k)log0.1$$

$$= -log0.1$$

### 4.3 Implementing the gradient of loss function for softmax regression (naive version)

The code of the gradient of loss function(naive version) for softmax regression is in Fig.5.

```
###############################################################################
# TODO: Compute the softmax loss and its gradient using explicit loops.      #
# Store the loss in J and the gradient in grad. If you are not               #
# careful here, it is easy to run into numeric instability. Don't forget     #
# the regularization term!                                                   #
###############################################################################

K=grad.shape[1]
for i in xrange(m):
  X_i =  X[i,:]
  score_i = np.dot(X_i,theta)
  s_max = -score_i.max()
  exp_score = np.exp(score_i+s_max)
  total_score = np.sum(exp_score , axis = 0)
  for k in xrange(K):
    grad[:,k] = grad[:,k] -X_i*(y[i]==k) + (exp_score[k] / total_score) * X_i
  numerator = np.exp(score_i[y[i]]+s_max)
  denom = np.sum(np.exp(score_i+s_max),axis = 0)
  J = J -np.log(numerator / float(denom))

J = J / float(m) + 0.5 * reg * np.sum(theta**2) / float(m)
grad = grad / float(m) + reg * theta / float(m)

###############################################################################
#                             END OF YOUR CODE                                #
###############################################################################
```

Fig.5 The gradient of loss function(naive version)

- **Checking your gradient implementation with numerical gradients**

By using the method of finite differences, we can check the implementation of the gradient of loss function(naive version) against the numerically approximated gradient. The result is in Fig.6.

```
numerical: 2.769691 analytic: 2.769691, relative error: 9.789110e-09
numerical: 1.820697 analytic: 1.820697, relative error: 1.515677e-09
numerical: -3.901717 analytic: -3.901717, relative error: 1.366744e-08
numerical: 0.809232 analytic: 0.809232, relative error: 1.184073e-08
numerical: -5.195859 analytic: -5.195859, relative error: 3.478235e-09
numerical: -2.738136 analytic: -2.738136, relative error: 3.250883e-08
numerical: -0.485431 analytic: -0.485431, relative error: 4.272427e-09
numerical: -0.282233 analytic: -0.282233, relative error: 6.545069e-08
numerical: -1.154074 analytic: -1.154074, relative error: 2.234206e-08
numerical: -1.892849 analytic: -1.892849, relative error: 3.094552e-09
naive loss: 2.369691e+00 computed in 36.544000s
```

Fig.6 The dfference between two gradients

From the Fig.6, we can see that the relative error between the two gradients of the order of $10^{-7}$ or less. So the gradient of loss function(naive version) we get is correct.

### 4.4 Implementing the loss function for softmax regression (vectorized version)

The code of the loss function(vectorized version) for softmax regression is in Fig.7.

```
    score_i = np.dot(theta.T,X.T)
    s_max = -np.max(score_i,axis=0)
    exp_score = np.exp(score_i+s_max)

    theta_new=theta[:,y]
    J=np.sum(np.multiply(theta_new,X.T)/float(m))
    J=J-np.sum((np.log(np.sum(exp_score,axis=0))-s_max)/float(m))
#    J=J-np.sum(np.log(np.sum(np.exp(score_i),axis=0))/float(m))
    J=-1.0*J+0.5*reg*np.sum(theta**2)/float(m)
```

Fig.7 The loss function(vectorized version)

### 4.5 Implementing the gradient of loss for softmax regression (vectorized version)

The code of the gradient of loss function(vectorized version) for softmax regression is in Fig.8.

```
K=grad.shape[1]
temp_1=exp_score
temp_2=np.sum(exp_score,axis=0)
temp=np.divide(temp_1,temp_2)
Grad=temp.T
Grad[np.arange(m),y] += -1.0
grad = np.dot(X.T,Grad) / float(m) + reg*theta/float(m)
```

Fig.8 The gradient of loss function(vectorized version)

• **Checking the implement of vectorized version** The difference between the result of the naive implement and the vectorized implement is in Fig.9.

```
vectorized loss: 2.369691e+00 computed in 5.537000s
Loss difference: 0.000000
Gradient difference: 0.000000
```

Fig.9 The dfference between two implements

From Fig.9, we can see that the relative error of the loss is 0, and the relative error of the gradient is 0 as well. So the implement of vectorized version is correct.

### 4.6 Implementing mini-batch gradient descent

The code of getting the batch is in Fig.10.

```
#########################################################################
# TODO:                                                                 #
# Sample batch_size elements from the training data and their           #
# corresponding labels to use in this round of gradient descent.        #
# Store the data in X_batch and their corresponding labels in           #
# y_batch; after sampling X_batch should have shape (batch_size, dim)   #
# and y_batch should have shape (batch_size,)                           #
#                                                                       #
# Hint: Use np.random.choice to generate indices. Sampling with         #
# replacement is faster than sampling without replacement.              #
# About 3 lines of code expected                                        #
#########################################################################
index=np.random.choice(num_train,batch_size)
X_batch=X[index,:]
y_batch=y[index]
#########################################################################
#                       END OF YOUR CODE                                #
#########################################################################
```

Fig.10 Getting the batch

The code of updating the parameter $\theta$ is in Fig.11.

```
#########################################################################
# TODO:                                                                 #
# Update the weights using the gradient and the learning rate.          #
# Hint: one line of code expected                                       #
#########################################################################
self.theta=self.theta-learning_rate*grad
#########################################################################
#                       END OF YOUR CODE                                #
#########################################################################
```

Fig.11 Updating the $\theta$

### 4.7 Using a validation set to select regularization lambda and learning rate for gradient descent

The code of selecting regularization lambda and learning rate for gradient descent is in Fig.12.

```
best_lamda=learning_rates[0]
best_reg=regularization_strengths[0]
from linear_classifier import LinearClassifier
for i in range(len(learning_rates)):
    for j in range(len(regularization_strengths)):
        lin=LinearClassifier()
        lin.train(X_train,y_train,learning_rates[i],regularization_strengths[j],4000,400,verbose=False)
        y_train_pred = lin.predict(X_train)
        accuracy_train = np.mean(y_train == y_train_pred)
        y_val_pred = lin.predict(X_val)
        accuracy_val = np.mean(y_val == y_val_pred)

        results[(learning_rates[i], regularization_strengths[j])] = (accuracy_train, accuracy_val)

        if (accuracy_val>best_val):
            best_val=accuracy_val
            best_softmax=lin
            best_lamda=learning_rates[i]
            best_reg=regularization_strengths[j]
```

Fig.12 Select regularization lambda and learning rate

We compute the gradient over batches of the training data, and use the validation set to compute the accuracy to choose the best values of the regularization lambda and the learning rate. The result is shown in Fig.13.

```
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.294020 val accuracy: 0.292000
lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.293816 val accuracy: 0.289000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.324408 val accuracy: 0.333000
lr 1.000000e-07 reg 1.000000e+08 train accuracy: 0.277653 val accuracy: 0.282000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.379041 val accuracy: 0.381000
lr 5.000000e-07 reg 1.000000e+05 train accuracy: 0.390388 val accuracy: 0.369000
lr 5.000000e-07 reg 5.000000e+05 train accuracy: 0.414000 val accuracy: 0.413000
lr 5.000000e-07 reg 1.000000e+08 train accuracy: 0.280796 val accuracy: 0.290000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.415306 val accuracy: 0.390000
lr 1.000000e-06 reg 1.000000e+05 train accuracy: 0.425959 val accuracy: 0.403000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.412878 val accuracy: 0.405000
lr 1.000000e-06 reg 1.000000e+08 train accuracy: 0.278673 val accuracy: 0.280000
lr 5.000000e-06 reg 5.000000e+04 train accuracy: 0.409612 val accuracy: 0.375000
lr 5.000000e-06 reg 1.000000e+05 train accuracy: 0.368449 val accuracy: 0.347000
lr 5.000000e-06 reg 5.000000e+05 train accuracy: 0.358510 val accuracy: 0.365000
lr 5.000000e-06 reg 1.000000e+08 train accuracy: 0.078714 val accuracy: 0.077000
best validation accuracy achieved during cross-validation: 0.413000
```

Fig.13 Select regularization lambda and learning rate

Here we choose a batch size of 400 and 4000 iterations for our experiment. And from Fig.13, we can get the best values of regularization lambda and learning rate with the bast validation accuracy. The best validation accuracy is 0.413, with the train accuracy 0.414, the regularization lambda 5e+05, and the learning rate 5e-07.

### 4.8 Training a softmax classier with the best hyperparameters

With the best values of regularization lambda and learning rate, We evaluate the test set accuracy.

Softmax on raw pixels final test set accuracy: 0.403400. The result of the confution matrix is

```
[[467  48  54  26  13  34  23  42 197  96]
 [ 54 483  17  30  18  36  49  38  99 176]
 [ 95  57 236  67 124  92 177  61  63  28]
 [ 36  85  87 207  48 206 138  63  60  70]
 [ 59  37 108  51 299  81 197  98  35  35]
 [ 33  46  99 116  64 354 114  70  72  32]
 [ 15  54  72  77  80  75 534  26  23  44]
 [ 50  57  61  38  95  78  67 399  53 102]
 [153  68   8  16   7  54  10  16 535 133]
 [ 68 169  13  23  12  17  49  45  84 520]]
```

• **Visualizing the learned parameter matrix** The result of the learned parameter matrix of softmax is in Fig.13.

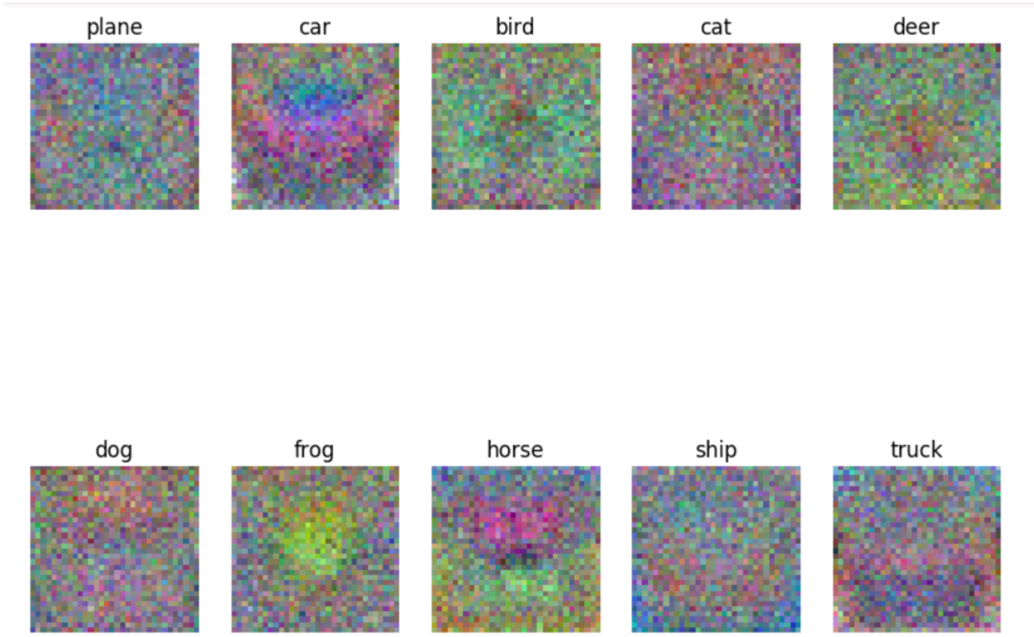


Fig.13 Visualizing the learned parameter matrix witha batch size of 400 and 4000 iterations

If we choose a batch size of 200 and 2000 iterations for our experiment, we can see that the picture is smoother. The result is in Fig.14. The test accuracy, however, is only 0.390800.

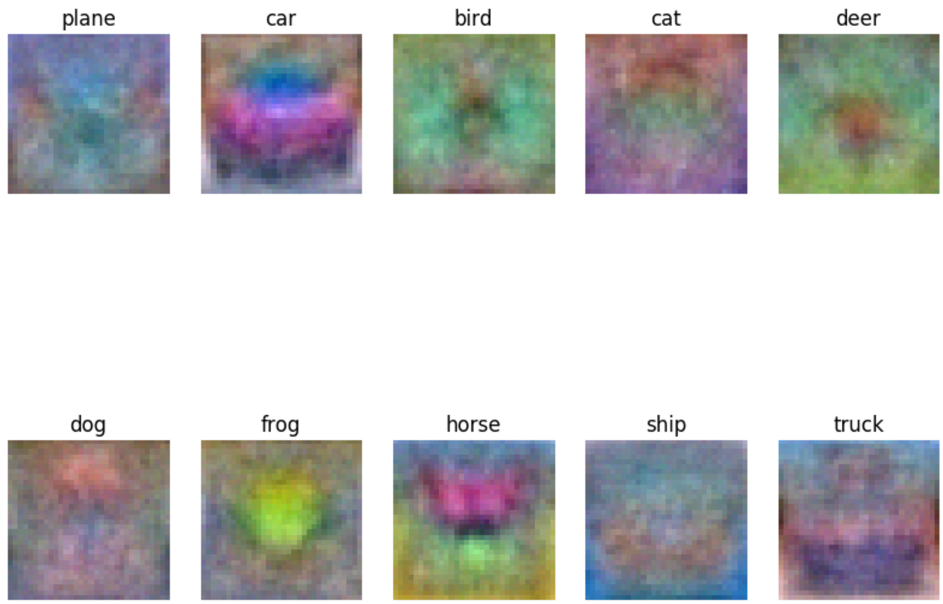


Fig.14 Visualizing the learned parameter matrix witha batch size of 200 and 2000 iterations

So, the more precise the visualizing result is, the more accuracy we'll get. That's the reason why a better test accuracy may accompany with a higher sharpness.

### 4.10 Comparing OVA binary logistic regression with softmax regression

For softmax, the performance is as follows.

When label=0:

| | $y = 0$ | otherwise |
|---|---|---|
| $\hat{y} = 0$ | 467 | 536 |
| otherwise | 563 | 3567 |

$$\Longrightarrow \begin{cases} specificity = 0.869 \\ accuracy = 0.786 \\ sensitivity = 0.453 \end{cases}$$

When label=1:

| | $y = 1$ | otherwise |
|---|---|---|
| $\hat{y} = 1$ | 483 | 517 |
| otherwise | 621 | 3551 |

$$\Longrightarrow \begin{cases} specificity = 0.873 \\ accuracy = 0.780 \\ sensitivity = 0.438 \end{cases}$$

When label=2:

| | $y=2$ | otherwise |
|---|---|---|
| $\hat{y}=2$ | 236 | 764 |
| otherwise | 519 | 3798 |

$$\Longrightarrow \begin{cases} specificity = 0.833 \\ accuracy = 0.759 \\ sensitivity = 0.313 \end{cases}$$

When label=3:

| | $y=3$ | otherwise |
|---|---|---|
| $\hat{y}=3$ | 207 | 793 |
| otherwise | 444 | 3827 |

$$\Longrightarrow \begin{cases} specificity = 0.828 \\ accuracy = 0.765 \\ sensitivity = 0.318 \end{cases}$$

When label=4:

| | $y=4$ | otherwise |
|---|---|---|
| $\hat{y}=4$ | 299 | 701 |
| otherwise | 461 | 3735 |

$$\Longrightarrow \begin{cases} specificity = 0.842 \\ accuracy = 0.776 \\ sensitivity = 0.393 \end{cases}$$

When label=5:

| | $y=5$ | otherwise |
|---|---|---|
| $\hat{y}=5$ | 354 | 646 |
| otherwise | 674 | 3680 |

$$\Longrightarrow \begin{cases} specificity = 0.851 \\ accuracy = 0.753 \\ sensitivity = 0.344 \end{cases}$$

When label=6:

| | $y=6$ | otherwise |
|---|---|---|
| $\hat{y}=6$ | 534 | 467 |
| otherwise | 824 | 3500 |

$$\Longrightarrow \begin{cases} specificity = 0.882 \\ accuracy = 0.758 \\ sensitivity = 0.393 \end{cases}$$

When label=7:

| | $y=7$ | otherwise |
|---|---|---|
| $\hat{y}=7$ | 399 | 601 |
| otherwise | 459 | 3635 |

$$\Longrightarrow \begin{cases} specificity = 0.858 \\ accuracy = 0.792 \\ sensitivity = 0.465 \end{cases}$$

When label=8:

| | $y=8$ | otherwise |
|---|---|---|
| $\hat{y}=8$ | 535 | 465 |
| otherwise | 686 | 3499 |

$$\Longrightarrow \begin{cases} specificity = 0.882 \\ accuracy = 0.778 \\ sensitivity = 0.438 \end{cases}$$

When label=9:

| | $y=9$ | otherwise |
|---|---|---|
| $\hat{y}=9$ | 520 | 480 |
| otherwise | 719 | 3514 |

$$\Longrightarrow \begin{cases} specificity = 0.880 \\ accuracy = 0.771 \\ sensitivity = 0.420 \end{cases}$$

So the table comparing the per genre classication performance of each classier is in Table.2.

| | y=0 | y=1 | y=2 | y=3 | y=4 | y=5 | y=6 | y=7 | y=8 | y=9 |
|---|---|---|---|---|---|---|---|---|---|---|
| specificity | 0.869 | 0.873 | 0.833 | 0.828 | 0.842 | 0.851 | 0.882 | 0.858 | 0.882 | 0.880 |
| accuracy | 0.786 | 0.780 | 0.759 | 0.765 | 0.775 | 0.753 | 0.758 | 0.792 | 0.778 | 0.771 |
| sensitivity | 0.453 | 0.438 | 0.313 | 0.318 | 0.393 | 0.344 | 0.393 | 0.465 | 0.438 | 0.420 |

Table.2 Per genre classication performance of each classier

The accuracy of both OVA and softmax are not very high. Their accuracy are all $<0.5$. So they are all not suitable to solve the problem.

From the difference between Table.1 and Table.2, we can see that the performance of softmax is better than OVA since specificity, accuracy and sensitivity of softmax are all higher than OVA.

So we recommend softmax for the CIFAR-10 classication problem.