svm

October 9, 2023

1

ipynb .

:

- SVM
-
-
-
- **SGD**
-

[1]:
```
#
import random
import numpy as np
from daseCV.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

#   notebook magic    matplotlib notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) #
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# magic     python
#       http://stackoverflow.com/questions/1907993/
↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.1 CIFAR-10

[2]:
```
#   CIFAR-10
cifar10_dir = 'daseCV/datasets/cifar-10-batches-py'

#
try:
   del X_train, y_train
```

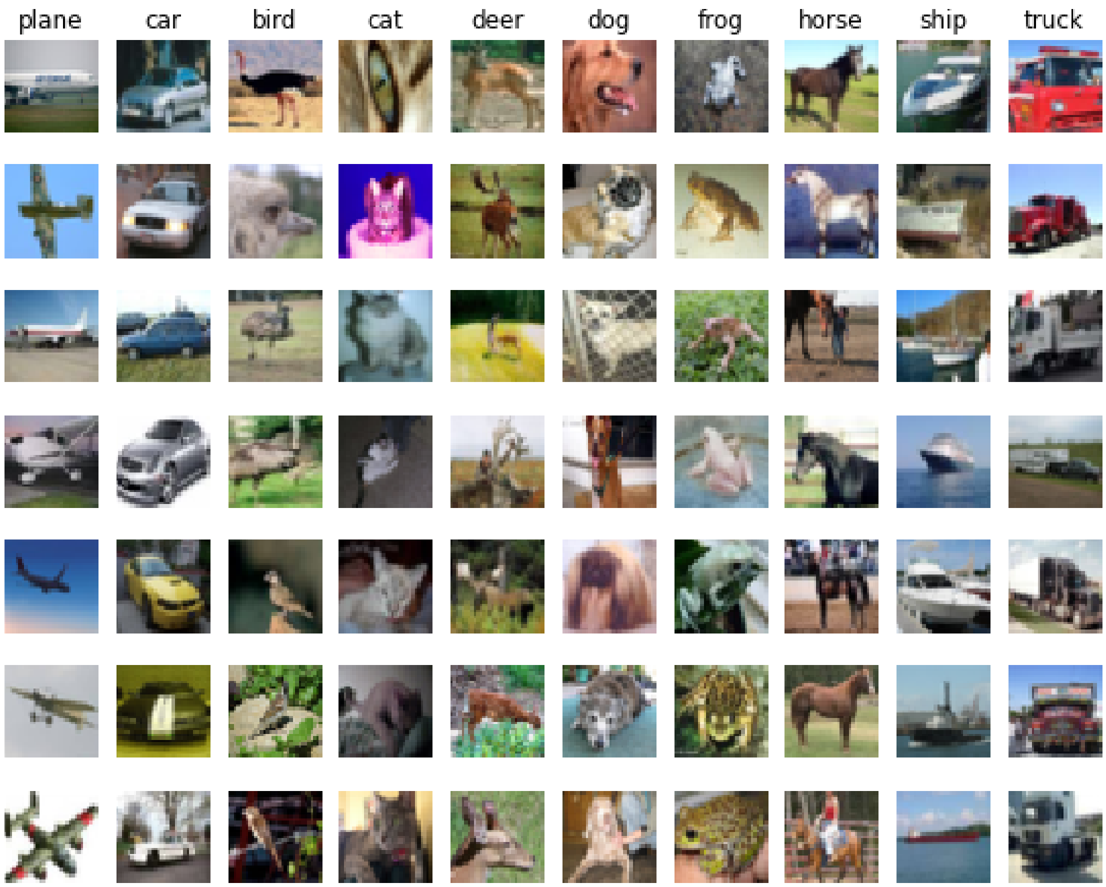
```
    del X_test, y_test
    print('Clear previously loaded data.')
except:
    pass

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

#
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)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[3]:
```
#
#       7
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[4]:

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

#       num_validation
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

#       num_training
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

#     num_dev
```

```
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

#       num_test
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
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,)
```

[5]:
```
#       rehspae
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))

#        shape
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

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

[6]:
```
#     image
#
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
↪image
```

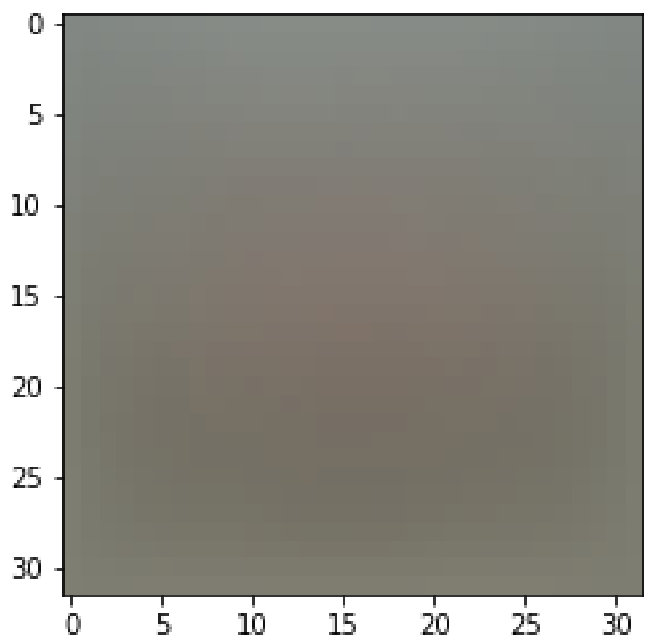
```
plt.show()

#
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

#     bias     1 bias trick
# SVM     bias       W
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))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```




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

1.2 SVM

daseCV/classifiers/linear_svm.py

`compute_loss_naive`   SVM

[7]:
```python
#
from daseCV.classifiers.linear_svm import svm_loss_naive
import time

#     SVM
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.091021
```

grad svm_loss_naive

:

[8]:
```python
#

#    W
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

#
#              (    )
from daseCV.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

#
#           100 (    )
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: 28.808778 analytic: 28.808778, relative error: 2.389835e-11
numerical: -14.987524 analytic: -14.987524, relative error: 4.582478e-11
numerical: -2.928714 analytic: -2.928714, relative error: 6.116952e-11
numerical: 4.229336 analytic: 4.229336, relative error: 2.443168e-11
numerical: -4.334515 analytic: -4.334515, relative error: 1.131135e-11
numerical: -4.428523 analytic: -4.428523, relative error: 6.186762e-11
numerical: 19.594190 analytic: 19.594190, relative error: 1.402505e-11
numerical: 23.967339 analytic: 23.967339, relative error: 1.320283e-11
numerical: 0.006630 analytic: 0.006630, relative error: 4.803128e-08
numerical: -8.494158 analytic: -8.494158, relative error: 1.058266e-11
numerical: 21.458136 analytic: 21.457694, relative error: 1.031862e-05
numerical: 7.821551 analytic: 7.830172, relative error: 5.508201e-04
numerical: 17.606607 analytic: 17.606139, relative error: 1.327319e-05
numerical: 16.143470 analytic: 16.144577, relative error: 3.428760e-05
numerical: 11.604099 analytic: 11.605755, relative error: 7.135526e-05
numerical: 15.424791 analytic: 15.426409, relative error: 5.244751e-05
```

```
numerical: 13.819052 analytic: 13.818000, relative error: 3.804799e-05
numerical: -16.623138 analytic: -16.624911, relative error: 5.332273e-05
numerical: 7.100931 analytic: 7.100874, relative error: 4.015153e-06
numerical: 2.352314 analytic: 2.361143, relative error: 1.873125e-03
```

1

gradcheck *SVM*

[9]:
```
#    svm_loss_vectorized
#
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from daseCV.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

#
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 9.091021e+00 computed in 0.903947s
Vectorized loss: 2.729088e-03 computed in 0.004694s
difference: 9.088292
```

[10]:
```
#  svm_loss_vectorized

#
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

#
#         Frobenius
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 1.078931s
```

```
Vectorized loss and gradient: computed in 0.004206s
difference: 0.000000
```

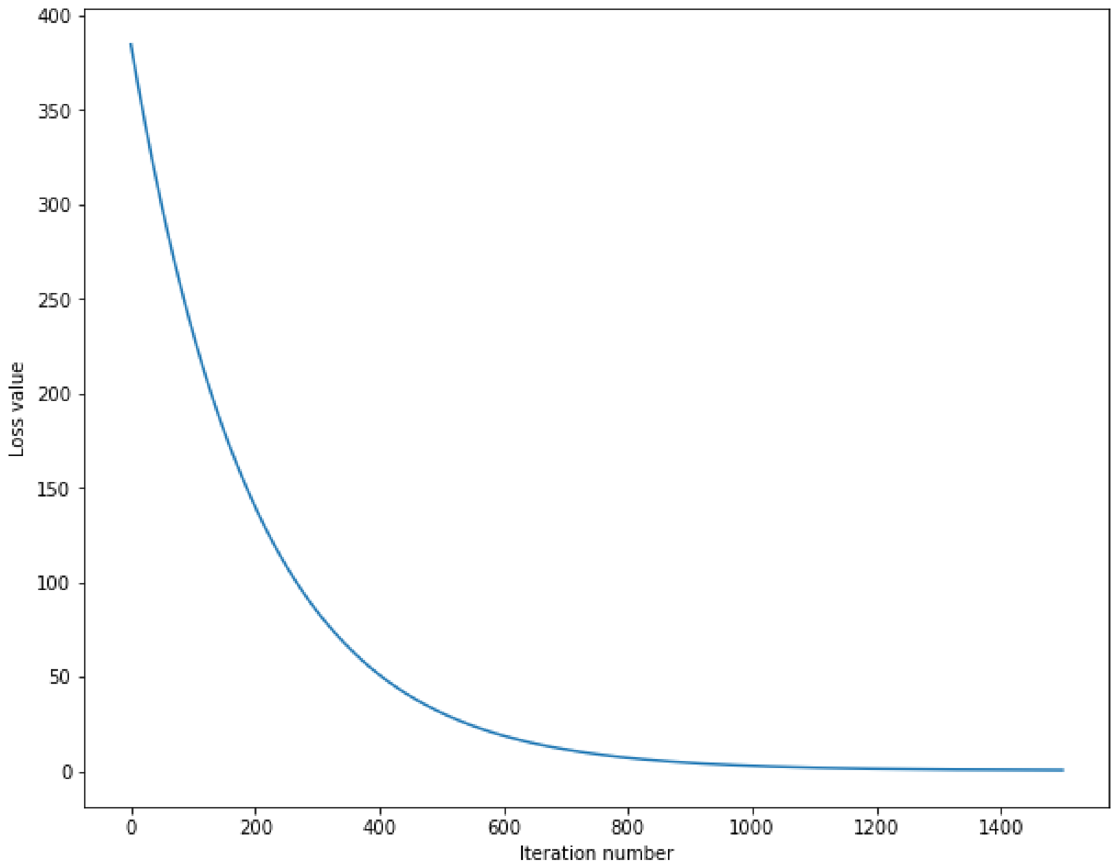
1.2.1 Stochastic Gradient Descent

SGD

[11]:
```python
# linear_classifier.py    LinearClassifier.train() SGD
#
from daseCV.classifiers import LinearSVM
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 384.669302
iteration 100 / 1500: loss 231.669048
iteration 200 / 1500: loss 139.723569
iteration 300 / 1500: loss 84.314496
iteration 400 / 1500: loss 50.930987
iteration 500 / 1500: loss 30.825557
iteration 600 / 1500: loss 18.710258
iteration 700 / 1500: loss 11.423748
iteration 800 / 1500: loss 7.031885
iteration 900 / 1500: loss 4.388581
iteration 1000 / 1500: loss 2.796281
iteration 1100 / 1500: loss 1.832050
iteration 1200 / 1500: loss 1.253337
iteration 1300 / 1500: loss 0.904085
iteration 1400 / 1500: loss 0.699356
That took 42.311926s
```

[12]:
```python
#    debugging
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



[13]:
```
#  LinearSVM.predict ,
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), ))
```

```
training accuracy: 0.383449
validation accuracy: 0.372000
```

[14]:
```
#       (     )
#            ;
#     ,        0.39

#   :        runtime/overflow
#       ,   bug

learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]
```

```python
# results    ,  (learning_rate, regularization_strength)   (training_accuracy, 
↪validation_accuracy)
# accuracy
results = {}
best_val = -1   #
best_svm = None #       LinearSVM

################################################################################
# TODO
#
#             SVM
#       results     best_val
#  best_svm     SVM
#
#   :
#           num_iter  SVM      ;
#               num_iter
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for i in learning_rates:
    for j in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=i, reg=j,
                              num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)

        results[(i, j)] = (train_acc, val_acc)

        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

#  results
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)
```

```
iteration 0 / 1500: loss 384.925630
```

```
iteration 100 / 1500: loss 231.791004
iteration 200 / 1500: loss 139.817195
iteration 300 / 1500: loss 84.388848
iteration 400 / 1500: loss 50.980964
iteration 500 / 1500: loss 30.852004
iteration 600 / 1500: loss 18.734792
iteration 700 / 1500: loss 11.428768
iteration 800 / 1500: loss 7.030957
iteration 900 / 1500: loss 4.385584
iteration 1000 / 1500: loss 2.799482
iteration 1100 / 1500: loss 1.839085
iteration 1200 / 1500: loss 1.258158
iteration 1300 / 1500: loss 0.907791
iteration 1400 / 1500: loss 0.696702
iteration 0 / 1500: loss 758.402796
iteration 100 / 1500: loss 276.417595
iteration 200 / 1500: loss 100.935752
iteration 300 / 1500: loss 37.033599
iteration 400 / 1500: loss 13.759277
iteration 500 / 1500: loss 5.297032
iteration 600 / 1500: loss 2.225938
iteration 700 / 1500: loss 1.089195
iteration 800 / 1500: loss 0.691439
iteration 900 / 1500: loss 0.538882
iteration 1000 / 1500: loss 0.489748
iteration 1100 / 1500: loss 0.469923
iteration 1200 / 1500: loss 0.467858
iteration 1300 / 1500: loss 0.458157
iteration 1400 / 1500: loss 0.459536
iteration 0 / 1500: loss 383.357618
iteration 100 / 1500: loss 394.677614
iteration 200 / 1500: loss 357.751055
iteration 300 / 1500: loss 410.400259
iteration 400 / 1500: loss 462.277554
iteration 500 / 1500: loss 475.746611
iteration 600 / 1500: loss 311.926577
iteration 700 / 1500: loss 452.296988
iteration 800 / 1500: loss 559.178589
iteration 900 / 1500: loss 471.596314
iteration 1000 / 1500: loss 558.981964
iteration 1100 / 1500: loss 552.652249
iteration 1200 / 1500: loss 340.822754
iteration 1300 / 1500: loss 468.203466
iteration 1400 / 1500: loss 412.556545
iteration 0 / 1500: loss 759.251156
iteration 100 / 1500: loss 36566842688439009569803605704538744 4224.000000
iteration 200 / 1500: loss 60442424946420824044156079818697590742607086144043641
296891964376855085056.000000
```

```
iteration 300 / 1500: loss 999058483589553119526884684969807994356333273929838001
2857136233168467065546211581020079561392448121025331200.000000
iteration 400 / 1500: loss 165133030755758340683486897947255796657295992522581495
5299269831078652655212797149757570246571086324611557828943307611220995547894534
606596079616.000000
iteration 500 / 1500: loss 272959315997748373687120209590585847873257515048808338
1955273378758743663604279873542132517662418523581643097238612245996163548115173
38639750980473409744959543519952971921264476160.000000
iteration 600 / 1500: loss 451176098995326568657627014349062119250533640633798903
4617391370642229308090435530057162192712080346309332173236611061290951346227387
90777378772198974862844054330698325791155480426662273288585948305739105344128286
72.000000
iteration 700 / 1500: loss 745754527381843918397965116399111075237218716025181851
0914127965067913103569229820436606710669884138910285516542522508417332604627528
25273299697647660102337806206298549297077667648498601807307628717412670963749566
3671052661078157156012085588625719296.000000
iteration 800 / 1500: loss 123268532460289927883958872873239514524885347392572395
0497915475997877986481327870694695228101699006167014615789325655068766295741529
81826369072980726959455825128617168245732790618938011249368435300725493516441421
0642125221968706021173226406976886599360308660091378052520454395615248384.000000

/home/public/10215501437-838-161/daseCV/classifiers/linear_svm.py:94:
RuntimeWarning: overflow encountered in double_scalars
  loss += 0.5 * reg * np.sum(W * W)
/opt/conda/lib/python3.9/site-packages/numpy/core/_methods.py:47:
RuntimeWarning: overflow encountered in reduce
  return umr_sum(a, axis, dtype, out, keepdims, initial, where)
/opt/conda/lib/python3.9/site-packages/numpy/core/fromnumeric.py:87:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/home/public/10215501437-838-161/daseCV/classifiers/linear_svm.py:92:
RuntimeWarning: invalid value encountered in multiply
  margin = (margin > 0) * np.sum(W * W)
/home/public/10215501437-838-161/daseCV/classifiers/linear_svm.py:92:
RuntimeWarning: overflow encountered in multiply
  margin = (margin > 0) * np.sum(W * W)
/home/public/10215501437-838-161/daseCV/classifiers/linear_svm.py:94:
RuntimeWarning: overflow encountered in multiply
  loss += 0.5 * reg * np.sum(W * W)

iteration 900 / 1500: loss nan
iteration 1000 / 1500: loss nan
iteration 1100 / 1500: loss nan
iteration 1200 / 1500: loss nan
iteration 1300 / 1500: loss nan
iteration 1400 / 1500: loss nan
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.386327 val accuracy: 0.393000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.369939 val accuracy: 0.377000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.082061 val accuracy: 0.084000
```

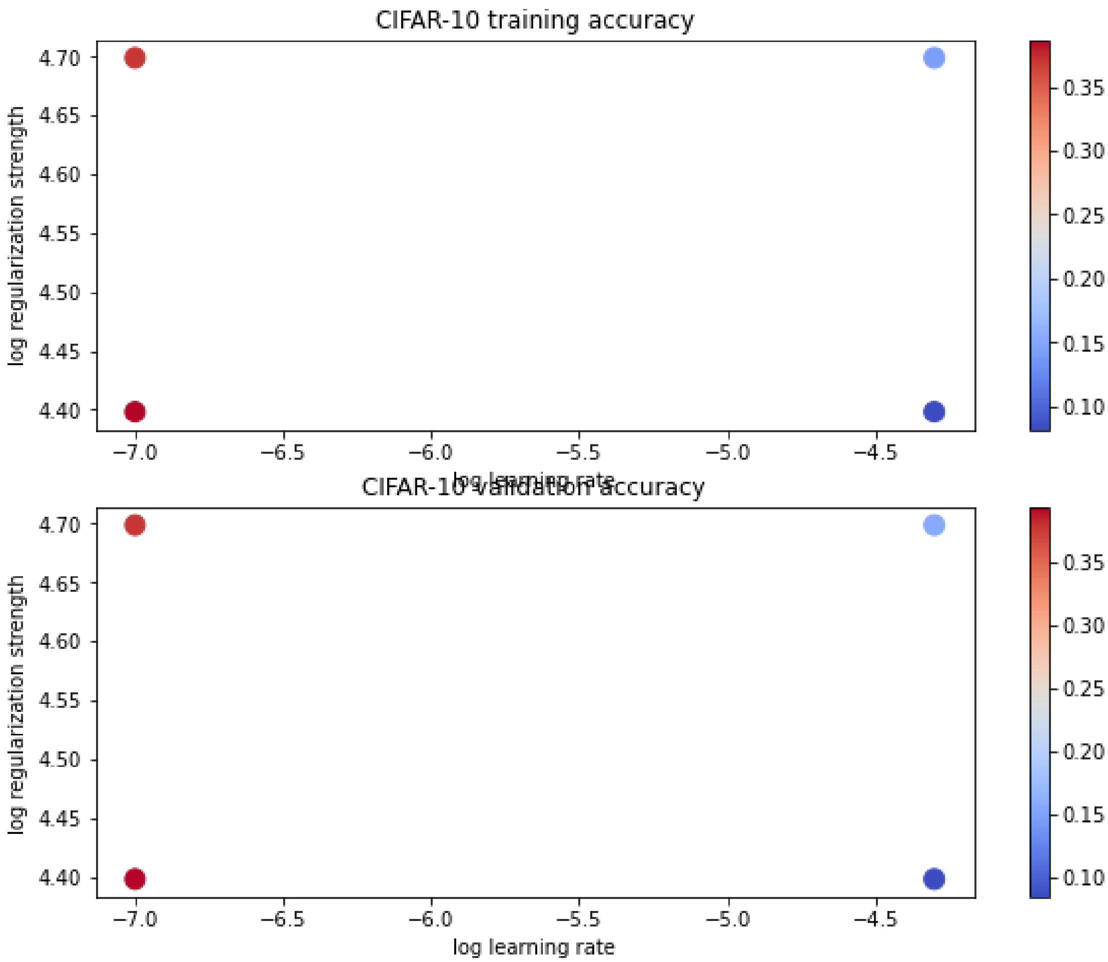
```
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.143551 val accuracy: 0.156000
best validation accuracy achieved during cross-validation: 0.393000
```

[15]:

```
#
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

#
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

#
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

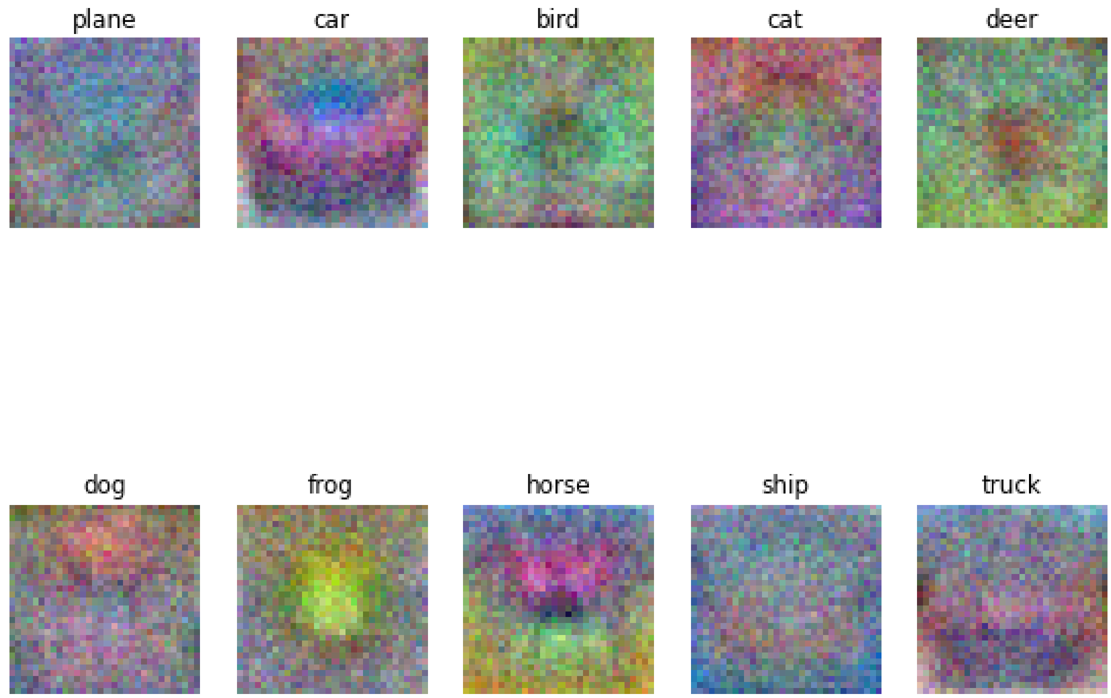


[16]:
```
#       SVM
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

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

[17]:
```
#
#
w = best_svm.W[:-1,:] #  bias
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)
```

```python
    #     0 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



2

SVM

1.3 Data for leaderboard

X leaderborad

---

[18]:
```python
# leaderboard
X = np.load("./input/X_3073.npy")
################################################################################
#     :
#     svm
#                   best_svm
################################################################################
```

```python
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
svm_leaderboard = best_svm
preds = svm_leaderboard.predict(X)
```

submit leaderboard phase2 leaderboard

[19]:
```python
import os
#
def output_file(preds, phase_id=2):
    path=os.getcwd()
    if not os.path.exists(path + '/output/phase_{}'.format(phase_id)):
        os.mkdir(path + '/output/phase_{}'.format(phase_id))
    path=path + '/output/phase_{}/prediction.npy'.format(phase_id)
    np.save(path,preds)
def zip_fun(phase_id=2):
    path=os.getcwd()
    output_path = path + '/output'
    files = os.listdir(output_path)
    for _file in files:
        if _file.find('zip') != -1:
            os.remove(output_path + '/' + _file)
    newpath=path+'/output/phase_{}'.format(phase_id)
    os.chdir(newpath)
    cmd = 'zip ../prediction_phase_{}.zip prediction.npy'.format(phase_id)
    os.system(cmd)
    os.chdir(path)
output_file(preds)
zip_fun()
```