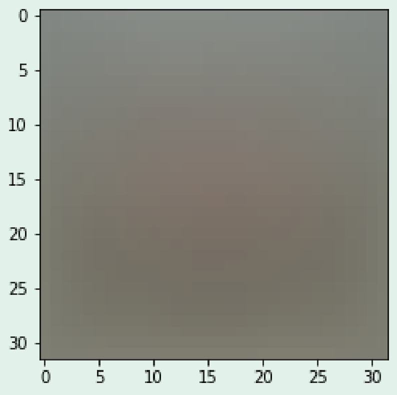
Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

```
[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)
```

SVM Classifier

Your code for this section will all be written inside **cs231n/classifiers/linear_svm.py**.

As you can see, we have prefilled the function `compute_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

In [7]:

```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
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: 8.855219
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

In [12]:

```
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should match
# almost exactly along all dimensions.
from cs231n.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)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
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: -5.614316 analytic: -5.614316, relative error: 2.136001e-11
numerical: 3.915729 analytic: 3.915729, relative error: 8.582426e-13
numerical: -9.291898 analytic: -9.291898, relative error: 9.006908e-12
numerical: 19.063197 analytic: 19.063197, relative error: 1.117650e-11
numerical: -34.205150 analytic: -34.205150, relative error: 4.069945e-12
numerical: -30.054175 analytic: -29.979092, relative error: 1.250697e-03
numerical: -5.614316 analytic: -5.614316, relative error: 2.136001e-11
numerical: -14.295408 analytic: -14.274346, relative error: 7.372315e-04
numerical: 17.122491 analytic: 17.122491, relative error: 2.221451e-11
numerical: 1.135152 analytic: 1.135152, relative error: 5.424631e-10
numerical: -9.357989 analytic: -9.357989, relative error: 2.026877e-11
numerical: 4.358376 analytic: 4.358376, relative error: 6.973882e-12
numerical: 6.601013 analytic: 6.601013, relative error: 8.083296e-11
numerical: 4.024407 analytic: 4.024407, relative error: 1.108147e-11
numerical: -0.918634 analytic: -0.918634, relative error: 2.291302e-10
numerical: -6.571149 analytic: -6.571149, relative error: 2.137595e-11
numerical: -50.784805 analytic: -50.784805, relative error: 4.555410e-12
numerical: 5.850735 analytic: 5.850735, relative error: 5.973168e-11
numerical: 23.451292 analytic: 23.451292, relative error: 1.463956e-12
numerical: -38.818530 analytic: -38.813444, relative error: 6.550628e-05
```

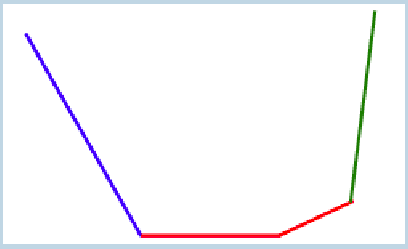
Inline Question 1

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

$YourAnswer:$ *fill this in.*

Yes, it is possible that a dimension in the gradcheck will not match exactly.

Such discrepancy may be caused by non-differentiable loss functions.



For example, in the image above (copied from lecture 3 optimization note), the kinks in the loss function (due to the max operation) technically make the loss function non-differentiable because at these kinks the gradient is not defined. At that time, numerical solution and analytical solution will be different.

In [13]:

```
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
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 cs231n.classifiers.linear_svm import svm_loss_vectorized
from cs231n.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))

# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

```
Naive loss: 8.855219e+00 computed in 0.082196s
Vectorized loss: 8.855219e+00 computed in 0.008733s
difference: 0.000000
```

In [14]:

```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
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))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.083400s
Vectorized loss and gradient: computed in 0.009060s
difference: 0.000000
```

Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss.

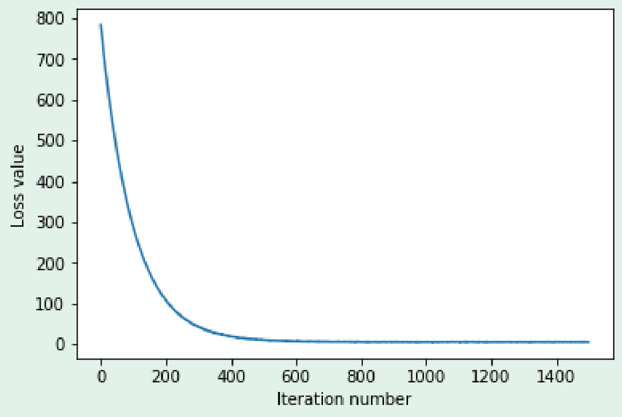
In [15]:

```
# In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from cs231n.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 783.192170
iteration 100 / 1500: loss 286.241058
iteration 200 / 1500: loss 107.962695
iteration 300 / 1500: loss 42.450987
iteration 400 / 1500: loss 19.047796
iteration 500 / 1500: loss 10.081456
iteration 600 / 1500: loss 7.199786
iteration 700 / 1500: loss 5.856792
iteration 800 / 1500: loss 5.269401
iteration 900 / 1500: loss 5.723489
iteration 1000 / 1500: loss 5.822409
iteration 1100 / 1500: loss 5.038547
iteration 1200 / 1500: loss 4.876252
iteration 1300 / 1500: loss 5.165719
iteration 1400 / 1500: loss 5.148642
That took 5.628599s
```

In [16]:

```
# A useful debugging strategy is to plot the loss as a function of
# A useful debugging strategy is to plot the loss as a function of
# iteration number:
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```



In [17]:

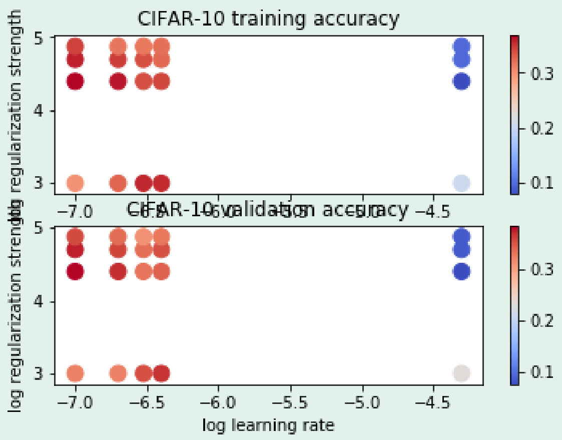
```
# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.368755
validation accuracy: 0.375000
```

In [19]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

#Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

learning_rates = [1e-7, 2e-7, 3e-7, 4e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4, 7.5e4, 1e3]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters as you develop your         #
# validation code so that the SVMs don't take much time to train; once you are #
# confident that your validation code works, you should rerun the validation   #
# code with a larger value for num_iters.                                      #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for rate in learning_rates:
    for strength in regularization_strengths:
        # Initialize a new svm every time, and train the data
        svm = LinearSVM()
        svm.train(X_train, y_train, rate, strength, num_iters=1000)
        y_train_pred = svm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)

        y_val_pred = svm.predict(X_val)
        validation_accuracy = np.mean(y_val == y_val_pred)

        # Store every result
        results[(rate, strength)] = (training_accuracy, validation_accuracy)

        # Store the best validation accuracy as the best_val
        if validation_accuracy > best_val:
            best_val = validation_accuracy
            best_svm = svm

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

# Print out 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)
```

```
lr 1.000000e-07 reg 1.000000e+03 train accuracy: 0.300347 val accuracy: 0.324000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.367143 val accuracy: 0.383000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.359837 val accuracy: 0.375000
lr 1.000000e-07 reg 7.500000e+04 train accuracy: 0.346327 val accuracy: 0.356000
lr 2.000000e-07 reg 1.000000e+03 train accuracy: 0.328469 val accuracy: 0.324000
lr 2.000000e-07 reg 2.500000e+04 train accuracy: 0.362571 val accuracy: 0.371000
lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.348571 val accuracy: 0.360000
lr 2.000000e-07 reg 7.500000e+04 train accuracy: 0.318592 val accuracy: 0.337000
lr 3.000000e-07 reg 1.000000e+03 train accuracy: 0.357878 val accuracy: 0.354000
lr 3.000000e-07 reg 2.500000e+04 train accuracy: 0.340163 val accuracy: 0.332000
lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.339449 val accuracy: 0.336000
lr 3.000000e-07 reg 7.500000e+04 train accuracy: 0.317531 val accuracy: 0.312000
lr 4.000000e-07 reg 1.000000e+03 train accuracy: 0.356673 val accuracy: 0.368000
lr 4.000000e-07 reg 2.500000e+04 train accuracy: 0.344143 val accuracy: 0.344000
lr 4.000000e-07 reg 5.000000e+04 train accuracy: 0.326082 val accuracy: 0.354000
lr 4.000000e-07 reg 7.500000e+04 train accuracy: 0.321306 val accuracy: 0.330000
lr 5.000000e-05 reg 1.000000e+03 train accuracy: 0.206163 val accuracy: 0.234000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.080163 val accuracy: 0.077000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
lr 5.000000e-05 reg 7.500000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.383000
```

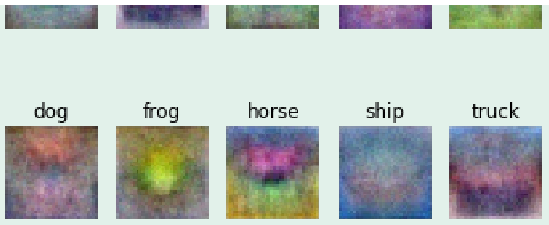


In [21]:

```
# Evaluate the best svm on test set
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: 0.370000
```

plane
car
bird
cat
deer



Inline question 2

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

$\color{blue}{Your Answer:}$ *fill this in*

The visualized SVM weights look like to have rough outlines of the corresponding objects but they are really unclear.

This is because the weights should present the basic features of a certain type of item but the same kind of items may have really different shapes and colors. So the SVM weight should also be relatively blurry in order to resemble different items that belong to the same class.