Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

In this exercise, you will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with 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

Acknowledgement: This exercise is adapted from Stanford CS231n.

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

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:
```
def rel_error(out, correct_out):
    return np.sum(abs(out - correct_out) / (abs(out) + abs(correct_out)))
```

In [3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    Softmax, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    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 = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # # We will also make a development set, which is a small subset of
    # the training set.
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    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))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    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))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data()
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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

In [4]:
```
# Create one-hot vectors for label
num_class = 10
y_train_oh = np.zeros((y_train.shape[0], 10))
y_train_oh[np.arange(y_train.shape[0]), y_train] = 1
y_val_oh = np.zeros((y_val.shape[0], 10))
y_val_oh[np.arange(y_val.shape[0]), y_val] = 1
y_test_oh = np.zeros((y_test.shape[0], 10))
y_test_oh[np.arange(y_test.shape[0]), y_test] = 1

y_dev_oh = np.zeros((y_dev.shape[0], 10))
y_dev_oh[np.arange(y_dev.shape[0]), y_dev] = 1
```

Regression as classifier

The most simple and straightforward approach to learn a classifier is to map the input data (raw image values) to class label (one-hot vector). The loss function is defined as following:

$$\mathcal{L} = \frac{1}{n}\|\mathbf{X}\mathbf{W} - \mathbf{y}\|_F^2 \qquad (1)$$

Where:

- $\mathbf{W} \in \mathbb{R}^{(d+1)\times C}$: Classifier weight
- $\mathbf{X} \in \mathbb{R}^{n\times(d+1)}$: Dataset
- $\mathbf{y} \in \mathbb{R}^{n\times C}$: Class label (one-hot vector)

Optimization

Given the loss function (1), the next problem is how to solve the weight $\mathbf{W}$. We now discuss 2 approaches:

- Random search
- Closed-form solution

Random search

In [5]:
```
bestloss = float('inf')
for num in range(100):
    W = np.random.randn(3073, 10) * 0.0001
    loss = np.linalg.norm(X_dev.dot(W) - y_dev_oh)
    if (loss < bestloss):
        bestloss = loss
        bestW = W
    print('in attempt %d the loss was %f, best %f' % (num, loss, bestloss))
```

```
in attempt 0 the loss was 34.584989, best 34.584989
in attempt 1 the loss was 32.742228, best 32.742228
in attempt 2 the loss was 33.160128, best 32.742228
in attempt 3 the loss was 33.415307, best 32.742228
in attempt 4 the loss was 30.138346, best 30.138346
in attempt 5 the loss was 34.948674, best 30.138346
in attempt 6 the loss was 32.965973, best 30.138346
in attempt 7 the loss was 34.918137, best 30.138346
in attempt 8 the loss was 32.294794, best 30.138346
in attempt 9 the loss was 32.126874, best 30.138346
in attempt 10 the loss was 31.166088, best 30.138346
in attempt 11 the loss was 34.075411, best 30.138346
in attempt 12 the loss was 31.195480, best 30.138346
in attempt 13 the loss was 32.638493, best 30.138346
in attempt 14 the loss was 37.145719, best 30.138346
in attempt 15 the loss was 35.623509, best 30.138346
in attempt 16 the loss was 32.539644, best 30.138346
in attempt 17 the loss was 32.865098, best 30.138346
in attempt 18 the loss was 33.973129, best 30.138346
in attempt 19 the loss was 32.522084, best 30.138346
in attempt 20 the loss was 32.957806, best 30.138346
in attempt 21 the loss was 33.585979, best 30.138346
in attempt 22 the loss was 32.507745, best 30.138346
in attempt 23 the loss was 34.235534, best 30.138346
in attempt 24 the loss was 32.319535, best 30.138346
in attempt 25 the loss was 33.779677, best 30.138346
in attempt 26 the loss was 32.288827, best 30.138346
in attempt 27 the loss was 33.289435, best 30.138346
in attempt 28 the loss was 32.868451, best 30.138346
in attempt 29 the loss was 32.106525, best 30.138346
in attempt 30 the loss was 33.924781, best 30.138346
in attempt 31 the loss was 32.749890, best 30.138346
in attempt 32 the loss was 37.061445, best 30.138346
in attempt 33 the loss was 30.424137, best 30.138346
in attempt 34 the loss was 34.672037, best 30.138346
in attempt 35 the loss was 33.579730, best 30.138346
in attempt 36 the loss was 35.456837, best 30.138346
in attempt 37 the loss was 33.319212, best 30.138346
in attempt 38 the loss was 32.360866, best 30.138346
in attempt 39 the loss was 36.350859, best 30.138346
in attempt 40 the loss was 34.209386, best 30.138346
in attempt 41 the loss was 34.008904, best 30.138346
in attempt 42 the loss was 33.750351, best 30.138346
in attempt 43 the loss was 34.551746, best 30.138346
in attempt 44 the loss was 33.940903, best 30.138346
in attempt 45 the loss was 34.247738, best 30.138346
in attempt 46 the loss was 32.590547, best 30.138346
in attempt 47 the loss was 35.363875, best 30.138346
in attempt 48 the loss was 35.369957, best 30.138346
in attempt 49 the loss was 33.035071, best 30.138346
in attempt 50 the loss was 37.049055, best 30.138346
in attempt 51 the loss was 32.885168, best 30.138346
in attempt 52 the loss was 31.628220, best 30.138346
in attempt 53 the loss was 33.244617, best 30.138346
in attempt 54 the loss was 34.467995, best 30.138346
in attempt 55 the loss was 34.363314, best 30.138346
in attempt 56 the loss was 35.917236, best 30.138346
in attempt 57 the loss was 32.522759, best 30.138346
in attempt 58 the loss was 32.768395, best 30.138346
in attempt 59 the loss was 35.309257, best 30.138346
in attempt 60 the loss was 33.347134, best 30.138346
in attempt 61 the loss was 33.450936, best 30.138346
in attempt 62 the loss was 32.831332, best 30.138346
in attempt 63 the loss was 32.640132, best 30.138346
in attempt 64 the loss was 32.810023, best 30.138346
in attempt 65 the loss was 34.843470, best 30.138346
in attempt 66 the loss was 33.290925, best 30.138346
in attempt 67 the loss was 36.720599, best 30.138346
in attempt 68 the loss was 33.134931, best 30.138346
in attempt 69 the loss was 32.255803, best 30.138346
in attempt 70 the loss was 32.640937, best 30.138346
in attempt 71 the loss was 32.591734, best 30.138346
in attempt 72 the loss was 33.997416, best 30.138346
in attempt 73 the loss was 34.428395, best 30.138346
in attempt 74 the loss was 31.865540, best 30.138346
in attempt 75 the loss was 34.075230, best 30.138346
in attempt 76 the loss was 32.786628, best 30.138346
in attempt 77 the loss was 33.204334, best 30.138346
in attempt 78 the loss was 33.501741, best 30.138346
in attempt 79 the loss was 33.501285, best 30.138346
in attempt 80 the loss was 32.429033, best 30.138346
in attempt 81 the loss was 33.193304, best 30.138346
in attempt 82 the loss was 33.659201, best 30.138346
in attempt 83 the loss was 35.416439, best 30.138346
in attempt 84 the loss was 33.040760, best 30.138346
in attempt 85 the loss was 33.297539, best 30.138346
in attempt 86 the loss was 34.237956, best 30.138346
in attempt 87 the loss was 31.215583, best 30.138346
in attempt 88 the loss was 32.738583, best 30.138346
in attempt 89 the loss was 34.129418, best 30.138346
in attempt 90 the loss was 33.061938, best 30.138346
in attempt 91 the loss was 33.802207, best 30.138346
in attempt 92 the loss was 32.481785, best 30.138346
in attempt 93 the loss was 34.712956, best 30.138346
in attempt 94 the loss was 33.498338, best 30.138346
in attempt 95 the loss was 37.406749, best 30.138346
in attempt 96 the loss was 32.674211, best 30.138346
in attempt 97 the loss was 32.395267, best 30.138346
in attempt 98 the loss was 33.646270, best 30.138346
in attempt 99 the loss was 32.294125, best 30.138346
```

In [6]:
```
# How bestW perform:
print('Accuracy on train set: ', np.sum(np.argmin(np.abs(1 - X_dev.dot(W)), axis=1) == y_dev).astype(np.float32
print('Accuracy on test set: ', np.sum(np.argmin(np.abs(1 - X_test.dot(W)), axis=1) == y_test).astype(np.float3
```

```
Accuracy on train set:  9.4
Accuracy on test set:  8.7
```

You can clearly see that the performance is very low, almost at the random level.

Closed-form solution

The closed-form solution is achieved by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\mathbf{W} - \mathbf{y}) = 0$$

$$\Leftrightarrow \mathbf{W}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

In [7]:
```
#########################################################################
# TODO:                                                                 #
# Implement the closed-form solution of the weight W.                   #
#########################################################################

A = np.matmul(X_train.T, X_train)
B = np.matmul(X_train.T, y_train_oh)
W = np.matmul(np.linalg.inv(A), B)

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

In [8]:
```
# Check accuracy:
print('Train set accuracy: ', np.sum(np.argmin(np.abs(1 - X_train.dot(W)), axis=1) == y_train).astype(np.float3
print('Test set accuracy: ', np.sum(np.argmin(np.abs(1 - X_test.dot(W)), axis=1) == y_test).astype(np.float32)/
```

```
Train set accuracy:  51.163265306122454
Test set accuracy:  36.199999999999996
```

Now, you can see that the performance is much better.

Regularization

A simple way to improve performance is to include the L2-regularization penalty.

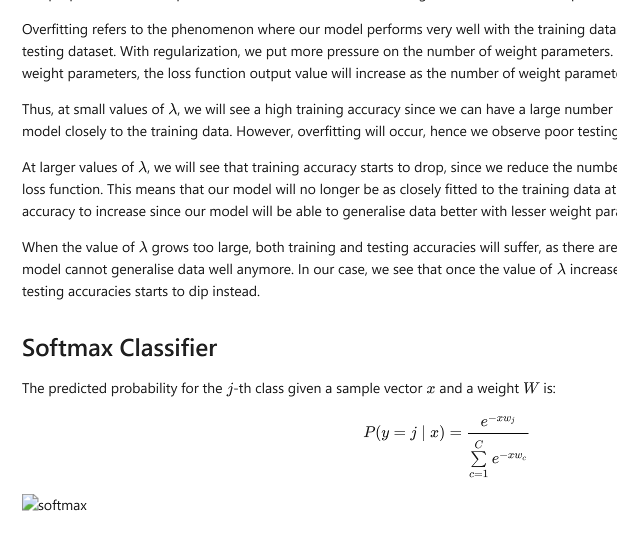
$$\mathcal{L} = \frac{1}{n}\|\mathbf{X}\mathbf{W} - \mathbf{y}\|_F^2 + \lambda\|\mathbf{W}\|_F^2 \qquad (2)$$

The closed-form solution now is:

$$\Leftrightarrow \mathbf{W}^* = (\mathbf{X}^T\mathbf{X} + \lambda n\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

In [9]:
```
# try several values of lambda to see how it helps:
lambdas = [0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
train_acc = np.zeros((len(lambdas)))
test_acc = np.zeros((len(lambdas)))
for i in range(len(lambdas)):
    l = lambdas[i]
    n,d = X_train.shape[0], X_train.shape[1]
    #########################################################################
    # TODO:                                                                 #
    # Implement the closed-form solution of the weight W with regularization.      #
    #########################################################################

    A = np.matmul(X_train.T, X_train) + (l * n * np.identity(d))
    B = np.matmul(X_train.T, y_train_oh)
    W = np.matmul(np.linalg.inv(A), B)

    #########################################################################
    #                          END OF YOUR CODE                             #
    #########################################################################
    train_acc[i] = np.sum(np.argmin(np.abs(1 - X_train.dot(W)), axis=1) == y_train).astype(np.float32)/y_train.
    test_acc[i]  = np.sum(np.argmin(np.abs(1 - X_test.dot(W)), axis=1) == y_test).astype(np.float32)/y_test.sha
```

In [10]:
```
plt.semilogx(lambdas, train_acc, 'r', label="Training accuracy")
plt.semilogx(lambdas, test_acc, 'g', label="Testing accuracy")

plt.legend()
plt.grid(True)
plt.show()
```



Question: Try to explain why the performances on the training and test set have such behaviors as we change the value of λ.

Your answer:
The purpose of have a λ parameter is to introduce the idea of regularization to solve the problem of overfitting.

Overfitting refers to the phenomenon where our model performs very well with the training dataset but experiences poor results with the testing dataset. With regularization, we put more pressure on the number of weight parameters. Therefore, by penalising the number of weight parameters, the loss function output value will increase as the number of weight parameters increase.

Thus, at small values of λ, we will see a high training accuracy since we can have a large number of weight parameters that helps to fit the model closely to the training data. However, overfitting will occur, hence we observe poor testing accuracy at this value of λ.

At larger values of λ, we will see that training accuracy starts to drop, since we reduce the number of weight parameters to minimise the loss function. This means that our model will no longer be as closely fitted to the training data at before. This will also cause the testing accuracy to increase since our model will be able to generalise data better with lesser weight parameters.

When the value of λ grows too large, both training and testing accuracies will suffer, as there are too little weight parameters and the model cannot generalise data well anymore. In our case, we see that once the value of λ increases beyond 10^3, both training and testing accuracies starts to dip instead.

Softmax Classifier

The predicted probability for the j-th class given a sample vector x and a weight W is:

$$P(y = j \mid x) = \frac{e^{-xw_j}}{\sum_{c=1}^{C} e^{-xw_c}}$$

softmax

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

In [11]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file classifiers/softmax.py and implement the
# softmax_loss_naive function.

from classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.337964
sanity check: 2.302585
```

Question: Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Your answer:
Since we are calculating our loss based on random weights (i.e., we haven't started the "learning" process yet), we expect that the initial loss has to be close to -log(0.1) because initially all the classes are equally likely to be chosen. Since CIFAR-10 consists of samples which belong to one of ten classes, the probability of the correct class will be 1/10 = 0.1. The softmax loss is the negative log probability of the correct class, therefore it is -log(0.1).

Optimization

Random search

In [12]:
```
bestloss = float('inf')
for num in range(100):
    W = np.random.randn(3073, 10) * 0.0001
    loss, _ = softmax_loss_naive(W, X_dev, y_dev, 0.0)
    if (loss < bestloss):
        bestloss = loss
        bestW = W
    print('in attempt %d the loss was %f, best %f' % (num, loss, bestloss))
```

```
in attempt 0 the loss was 2.358189, best 2.358189
in attempt 1 the loss was 2.381420, best 2.358189
in attempt 2 the loss was 2.335523, best 2.335523
in attempt 3 the loss was 2.310783, best 2.310783
in attempt 4 the loss was 2.353139, best 2.310783
in attempt 5 the loss was 2.392291, best 2.310783
in attempt 6 the loss was 2.362636, best 2.310783
in attempt 7 the loss was 2.365318, best 2.310783
in attempt 8 the loss was 2.416753, best 2.310783
in attempt 9 the loss was 2.364832, best 2.310783
in attempt 10 the loss was 2.358573, best 2.310783
in attempt 11 the loss was 2.353233, best 2.310783
in attempt 12 the loss was 2.400715, best 2.310783
in attempt 13 the loss was 2.346051, best 2.310783
in attempt 14 the loss was 2.364322, best 2.310783
in attempt 15 the loss was 2.339410, best 2.310783
in attempt 16 the loss was 2.340192, best 2.310783
in attempt 17 the loss was 2.363616, best 2.310783
in attempt 18 the loss was 2.359163, best 2.310783
in attempt 19 the loss was 2.369597, best 2.310783
in attempt 20 the loss was 2.388587, best 2.310783
in attempt 21 the loss was 2.392862, best 2.310783
in attempt 22 the loss was 2.313808, best 2.310783
in attempt 23 the loss was 2.423103, best 2.310783
in attempt 24 the loss was 2.359853, best 2.310783
in attempt 25 the loss was 2.364470, best 2.310783
in attempt 26 the loss was 2.374066, best 2.310783
in attempt 27 the loss was 2.317560, best 2.310783
in attempt 28 the loss was 2.359727, best 2.310783
in attempt 29 the loss was 2.388236, best 2.310783
in attempt 30 the loss was 2.356464, best 2.310783
in attempt 31 the loss was 2.417594, best 2.310783
in attempt 32 the loss was 2.373376, best 2.310783
in attempt 33 the loss was 2.387114, best 2.310783
in attempt 34 the loss was 2.362240, best 2.310783
in attempt 35 the loss was 2.331767, best 2.310783
in attempt 36 the loss was 2.365536, best 2.310783
in attempt 37 the loss was 2.367833, best 2.310783
in attempt 38 the loss was 2.278711, best 2.278711
in attempt 39 the loss was 2.355449, best 2.278711
in attempt 40 the loss was 2.390439, best 2.278711
in attempt 41 the loss was 2.357839, best 2.278711
in attempt 42 the loss was 2.321468, best 2.278711
in attempt 43 the loss was 2.326473, best 2.278711
in attempt 44 the loss was 2.364281, best 2.278711
in attempt 45 the loss was 2.394717, best 2.278711
in attempt 46 the loss was 2.322318, best 2.278711
in attempt 47 the loss was 2.400970, best 2.278711
in attempt 48 the loss was 2.359326, best 2.278711
in attempt 49 the loss was 2.272516, best 2.272516
in attempt 50 the loss was 2.310517, best 2.272516
in attempt 51 the loss was 2.365610, best 2.272516
in attempt 52 the loss was 2.365215, best 2.272516
in attempt 53 the loss was 2.381956, best 2.272516
in attempt 54 the loss was 2.401309, best 2.272516
in attempt 55 the loss was 2.387564, best 2.272516
in attempt 56 the loss was 2.322308, best 2.272516
in attempt 57 the loss was 2.360626, best 2.272516
in attempt 58 the loss was 2.310527, best 2.272516
in attempt 59 the loss was 2.383944, best 2.272516
in attempt 60 the loss was 2.347391, best 2.272516
in attempt 61 the loss was 2.317753, best 2.272516
in attempt 62 the loss was 2.330848, best 2.272516
in attempt 63 the loss was 2.328584, best 2.272516
in attempt 64 the loss was 2.414387, best 2.272516
in attempt 65 the loss was 2.307981, best 2.272516
in attempt 66 the loss was 2.383025, best 2.272516
in attempt 67 the loss was 2.367787, best 2.272516
in attempt 68 the loss was 2.326626, best 2.272516
in attempt 69 the loss was 2.327702, best 2.272516
in attempt 70 the loss was 2.391061, best 2.272516
in attempt 71 the loss was 2.384977, best 2.272516
in attempt 72 the loss was 2.351067, best 2.272516
in attempt 73 the loss was 2.345127, best 2.272516
in attempt 74 the loss was 2.384248, best 2.272516
in attempt 75 the loss was 2.400660, best 2.272516
in attempt 76 the loss was 2.330432, best 2.272516
in attempt 77 the loss was 2.344338, best 2.272516
in attempt 78 the loss was 2.330186, best 2.272516
in attempt 79 the loss was 2.314007, best 2.272516
in attempt 80 the loss was 2.366572, best 2.272516
in attempt 81 the loss was 2.377288, best 2.272516
in attempt 82 the loss was 2.343578, best 2.272516
in attempt 83 the loss was 2.346955, best 2.272516
in attempt 84 the loss was 2.425035, best 2.272516
in attempt 85 the loss was 2.326370, best 2.272516
in attempt 86 the loss was 2.423469, best 2.272516
in attempt 87 the loss was 2.342048, best 2.272516
in attempt 88 the loss was 2.411416, best 2.272516
in attempt 89 the loss was 2.389891, best 2.272516
in attempt 90 the loss was 2.319877, best 2.272516
in attempt 91 the loss was 2.385751, best 2.272516
in attempt 92 the loss was 2.341823, best 2.272516
in attempt 93 the loss was 2.425128, best 2.272516
in attempt 94 the loss was 2.384023, best 2.272516
in attempt 95 the loss was 2.351891, best 2.272516
in attempt 96 the loss was 2.343007, best 2.272516
in attempt 97 the loss was 2.313557, best 2.272516
in attempt 98 the loss was 2.326912, best 2.272516
in attempt 99 the loss was 2.345184, best 2.272516
```

In [13]:
```
# How bestW perform on trainset
scores = X_train.dot(bestW)
y_pred = np.argmax(scores, axis=1)
print('Accuracy on train set %f' % np.mean(y_pred == y_train))

# evaluate performance of test set
scores = X_test.dot(bestW)
y_pred = np.argmax(scores, axis=1)
print('Accuracy on test set %f' % np.mean(y_pred == y_test))
```

```
Accuracy on train set 0.132959
Accuracy on test set 0.158000
```

Compare the performance when using random search with *regression classifier* and *softmax classifier*. You can see how much useful the softmax classifier is.

Stochastic Gradient descent

Even though it is possible to achieve closed-form solution with softmax classifier, it would be more complicated. In fact, we could achieve very good results with gradient descent approach. Additionally, in case of very large dataset, it is impossible to load the whole dataset into the memory. Gradient descent can help to optimize the loss function in batch.

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \alpha\frac{\partial \mathcal{L}(\mathbf{x}; \mathbf{W}^t)}{\partial \mathbf{W}^t}$$

Where α is the learning rate, $\mathcal{L}$ is a loss function, and $\mathbf{x}$ is a batch of training dataset.

In [14]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# Use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 1e2)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 1e2)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -1.268520 analytic: -1.268520, relative error: 2.673774e-09
numerical: -1.498243 analytic: -1.498243, relative error: 4.546300e-09
numerical: -1.091292 analytic: -1.091292, relative error: 2.083992e-08
numerical: -1.282992 analytic: -1.282992, relative error: 2.987137e-08
numerical: -2.215568 analytic: -2.215568, relative error: 9.756394e-09
numerical: -2.071038 analytic: -2.071038, relative error: 1.663883e-08
numerical: 2.138951 analytic: 2.138951, relative error: 2.953210e-08
numerical: -6.087443 analytic: -6.087443, relative error: 2.667196e-09
numerical: 1.951866 analytic: 1.951866, relative error: 9.234415e-09
numerical: 2.643592 analytic: 2.643592, relative error: 8.611758e-09
numerical: 3.104153 analytic: 3.104153, relative error: 1.037942e-09
numerical: -3.464932 analytic: -3.464933, relative error: 2.571873e-08
numerical: 1.950068 analytic: 1.950068, relative error: 4.875809e-09
numerical: 3.118712 analytic: 3.118712, relative error: 3.211463e-08
numerical: -0.523393 analytic: -0.523393, relative error: 1.889307e-08
numerical: -0.842392 analytic: -0.842392, relative error: 1.106036e-07
numerical: 0.510743 analytic: 0.510743, relative error: 1.861022e-08
numerical: 2.391333 analytic: 2.391333, relative error: 3.636120e-08
numerical: 1.400442 analytic: 1.400442, relative error: 2.312027e-08
numerical: -0.925175 analytic: -0.925175, relative error: 4.953093e-08
```

In [17]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.00001)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.00001)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# We use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.345184e+00 computed in 0.084995s
vectorized loss: 2.345184e+00 computed in 0.004003s
Loss difference: 0.000000
Gradient difference: 0.000000
```

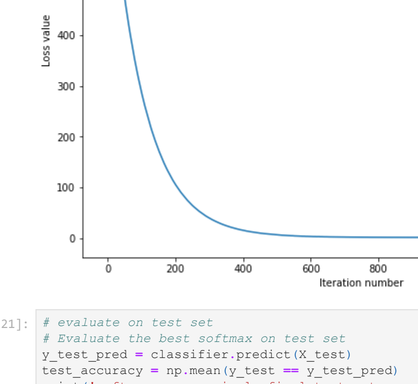
In [18]:
```
from classifiers.linear_classifier import *

classifier = Softmax()
tic = time.time()
loss_hist = classifier.train(X_train, y_train, learning_rate=1e-7, reg=5e4,
                             num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 779.794758
iteration 100 / 1500: loss 285.888821
iteration 200 / 1500: loss 106.043107
iteration 300 / 1500: loss 40.132818
iteration 400 / 1500: loss 16.034196
iteration 500 / 1500: loss 7.201730
iteration 600 / 1500: loss 3.971111
iteration 700 / 1500: loss 2.785901
iteration 800 / 1500: loss 2.359809
iteration 900 / 1500: loss 2.182554
iteration 1000 / 1500: loss 2.132032
iteration 1100 / 1500: loss 2.117795
iteration 1200 / 1500: loss 2.118989
iteration 1300 / 1500: loss 2.062584
iteration 1400 / 1500: loss 2.073079
That took 4.501000s
```

In [19]:
```
# Write the Softmax.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = classifier.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = classifier.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.334000
validation accuracy: 0.353000
```

In [20]:
```
# 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 [21]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = classifier.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %.2f' % (100*test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 35.50
```