softmax

February 21, 2019

1 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

```
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=5
            """
            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
```

2 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:

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

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)

3 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

3.1 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 34.393928, best 34.393928
in attempt 1 the loss was 30.519490, best 30.519490
in attempt 2 the loss was 32.449885, best 30.519490
in attempt 3 the loss was 32.358632, best 30.519490
in attempt 4 the loss was 34.428078, best 30.519490
in attempt 5 the loss was 32.923809, best 30.519490
in attempt 6 the loss was 33.867275, best 30.519490
in attempt 7 the loss was 35.006327, best 30.519490
in attempt 8 the loss was 33.986503, best 30.519490
in attempt 9 the loss was 33.752676, best 30.519490
in attempt 10 the loss was 31.852424, best 30.519490
in attempt 11 the loss was 32.867633, best 30.519490
in attempt 12 the loss was 31.901147, best 30.519490
in attempt 13 the loss was 30.909278, best 30.519490
in attempt 14 the loss was 33.618547, best 30.519490
in attempt 15 the loss was 32.378885, best 30.519490
in attempt 16 the loss was 31.833938, best 30.519490
in attempt 17 the loss was 32.677192, best 30.519490
in attempt 18 the loss was 33.661380, best 30.519490
in attempt 19 the loss was 32.035283, best 30.519490
in attempt 20 the loss was 35.076666, best 30.519490
in attempt 21 the loss was 32.462978, best 30.519490
in attempt 22 the loss was 31.696446, best 30.519490
in attempt 23 the loss was 31.480120, best 30.519490
in attempt 24 the loss was 36.665358, best 30.519490
in attempt 25 the loss was 32.805700, best 30.519490
in attempt 26 the loss was 33.464136, best 30.519490
in attempt 27 the loss was 34.996252, best 30.519490
in attempt 28 the loss was 32.723091, best 30.519490
in attempt 29 the loss was 31.078483, best 30.519490
in attempt 30 the loss was 32.546250, best 30.519490
in attempt 31 the loss was 33.521194, best 30.519490
in attempt 32 the loss was 32.155400, best 30.519490
in attempt 33 the loss was 35.270959, best 30.519490
in attempt 34 the loss was 33.013106, best 30.519490
in attempt 35 the loss was 31.290493, best 30.519490
in attempt 36 the loss was 32.312153, best 30.519490
in attempt 37 the loss was 32.962256, best 30.519490
in attempt 38 the loss was 31.642363, best 30.519490
in attempt 39 the loss was 35.561708, best 30.519490
in attempt 40 the loss was 35.363973, best 30.519490
in attempt 41 the loss was 32.375050, best 30.519490
```

```
in attempt 42 the loss was 32.683851, best 30.519490
in attempt 43 the loss was 34.630583, best 30.519490
in attempt 44 the loss was 31.882045, best 30.519490
in attempt 45 the loss was 32.836921, best 30.519490
in attempt 46 the loss was 33.509347, best 30.519490
in attempt 47 the loss was 36.003205, best 30.519490
in attempt 48 the loss was 35.554924, best 30.519490
in attempt 49 the loss was 34.641597, best 30.519490
in attempt 50 the loss was 32.906969, best 30.519490
in attempt 51 the loss was 31.972947, best 30.519490
in attempt 52 the loss was 32.304989, best 30.519490
in attempt 53 the loss was 31.422587, best 30.519490
in attempt 54 the loss was 32.506871, best 30.519490
in attempt 55 the loss was 31.682542, best 30.519490
in attempt 56 the loss was 34.803236, best 30.519490
in attempt 57 the loss was 32.551212, best 30.519490
in attempt 58 the loss was 33.464934, best 30.519490
in attempt 59 the loss was 33.894489, best 30.519490
in attempt 60 the loss was 32.240282, best 30.519490
in attempt 61 the loss was 32.287951, best 30.519490
in attempt 62 the loss was 35.498825, best 30.519490
in attempt 63 the loss was 31.046850, best 30.519490
in attempt 64 the loss was 31.346946, best 30.519490
in attempt 65 the loss was 33.803244, best 30.519490
in attempt 66 the loss was 33.476334, best 30.519490
in attempt 67 the loss was 35.990624, best 30.519490
in attempt 68 the loss was 34.088332, best 30.519490
in attempt 69 the loss was 33.609976, best 30.519490
in attempt 70 the loss was 32.766351, best 30.519490
in attempt 71 the loss was 32.456712, best 30.519490
in attempt 72 the loss was 33.468071, best 30.519490
in attempt 73 the loss was 33.780081, best 30.519490
in attempt 74 the loss was 30.974518, best 30.519490
in attempt 75 the loss was 32.908449, best 30.519490
in attempt 76 the loss was 31.539249, best 30.519490
in attempt 77 the loss was 31.900745, best 30.519490
in attempt 78 the loss was 33.930264, best 30.519490
in attempt 79 the loss was 31.750968, best 30.519490
in attempt 80 the loss was 33.317366, best 30.519490
in attempt 81 the loss was 33.215185, best 30.519490
in attempt 82 the loss was 35.833957, best 30.519490
in attempt 83 the loss was 32.584753, best 30.519490
in attempt 84 the loss was 33.044257, best 30.519490
in attempt 85 the loss was 32.778402, best 30.519490
in attempt 86 the loss was 34.649594, best 30.519490
in attempt 87 the loss was 32.702203, best 30.519490
in attempt 88 the loss was 32.955587, best 30.519490
in attempt 89 the loss was 33.083292, best 30.519490
```

```
in attempt 90 the loss was 31.945120, best 30.519490
in attempt 91 the loss was 31.118071, best 30.519490
in attempt 92 the loss was 33.106180, best 30.519490
in attempt 93 the loss was 34.260280, best 30.519490
in attempt 94 the loss was 31.927740, best 30.519490
in attempt 95 the loss was 32.694747, best 30.519490
in attempt 96 the loss was 33.475632, best 30.519490
in attempt 97 the loss was 33.433384, best 30.519490
in attempt 98 the loss was 32.492593, best 30.519490
in attempt 99 the loss was 34.629709, best 30.519490
```

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

```
Accuracy on train set:  8.200000000000001
Accuracy on test set:  11.700000000000001
```

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

3.2 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.                          #
        ################################################################################
        W = np.matmul(np.matmul(np.linalg.inv(np.matmul(X_train.T, X_train)),X_train.T), y_tra

        ################################################################################
        #                              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
        print('Test set accuracy: ', np.sum(np.argmin(np.abs(1 - X_test.dot(W)), axis=1) == y_t
```

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

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

3.3 Regularization

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

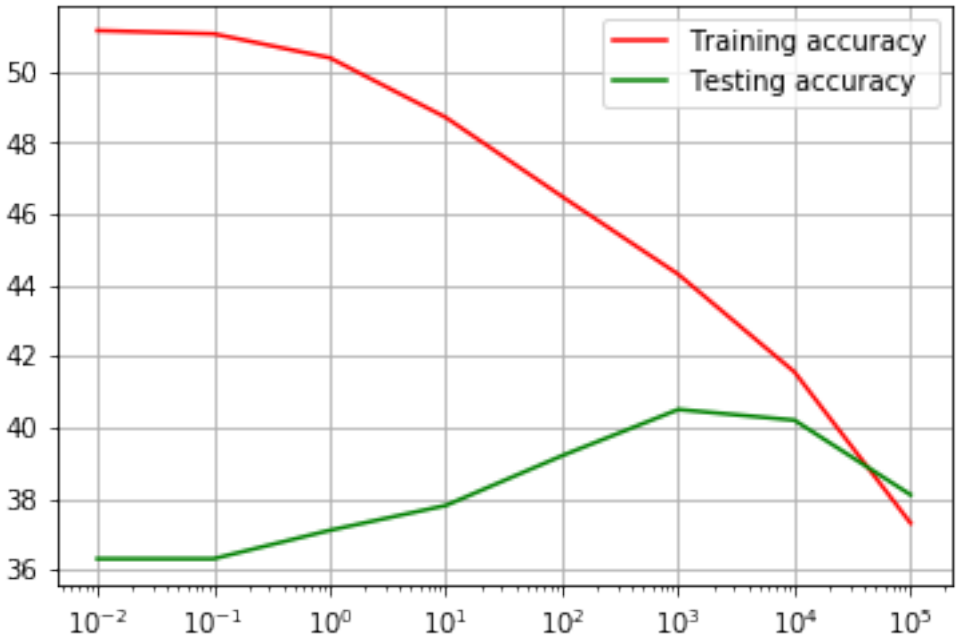
$$\mathcal{L} = \frac{1}{n}\|\mathbf{XW} - \mathbf{y}\|_F^2 + \lambda\|\mathbf{W}\|_F^2 \qquad (2)$$

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.      #
            ################################################################################
            W = np.matmul(np.matmul(np.linalg.inv(np.matmul(X_train.T,X_train) + n*l*np.identit
            ################################################################################
            #                                END OF YOUR CODE                              #
            ################################################################################
            train_acc[i] = np.sum(np.argmin(np.abs(1 - X_train.dot(W)), axis=1) == y_train).as
            test_acc[i]  = np.sum(np.argmin(np.abs(1 - X_test.dot(W)), axis=1) == y_test).asty
```

```
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: Fill in here

As regularization increases, training accuracy decreases since training (using the training data) is penalised. Thus the model becomes further away from the training data.

Testing accuracy increases then decrease as initially, the model becomes more "generalised" allowing better prediction on the testing set. However, as regularization increases further, it becomes overly regularized.

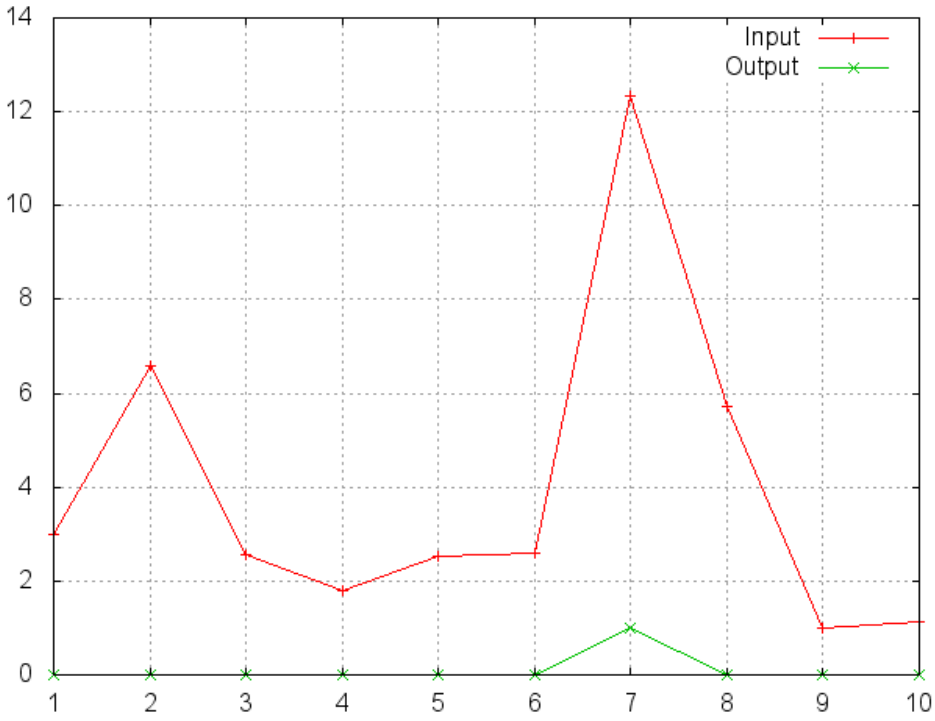
3.4 Softmax Classifier

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

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

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.
```



softmax

```
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: 2.365720
sanity check: 2.302585
```

Question: Why do we expect our loss to be close to -log(0.1)? Explain briefly.**
Your answer: Fill in here
There are 10 classes. It is the average probability per class

4 Optimization

4.1 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 2.444791, best 2.444791
in attempt 1 the loss was 2.350226, best 2.350226
in attempt 2 the loss was 2.359253, best 2.350226
in attempt 3 the loss was 2.355820, best 2.350226
in attempt 4 the loss was 2.406799, best 2.350226
in attempt 5 the loss was 2.362762, best 2.350226
in attempt 6 the loss was 2.333948, best 2.333948
in attempt 7 the loss was 2.314687, best 2.314687
in attempt 8 the loss was 2.327703, best 2.314687
in attempt 9 the loss was 2.297980, best 2.297980
in attempt 10 the loss was 2.349750, best 2.297980
in attempt 11 the loss was 2.402569, best 2.297980
in attempt 12 the loss was 2.350318, best 2.297980
in attempt 13 the loss was 2.383877, best 2.297980
in attempt 14 the loss was 2.359327, best 2.297980
in attempt 15 the loss was 2.352645, best 2.297980
in attempt 16 the loss was 2.330629, best 2.297980
in attempt 17 the loss was 2.316823, best 2.297980
in attempt 18 the loss was 2.330964, best 2.297980
in attempt 19 the loss was 2.345581, best 2.297980
in attempt 20 the loss was 2.382264, best 2.297980
in attempt 21 the loss was 2.341777, best 2.297980
in attempt 22 the loss was 2.352340, best 2.297980
in attempt 23 the loss was 2.464117, best 2.297980
in attempt 24 the loss was 2.330748, best 2.297980
in attempt 25 the loss was 2.307026, best 2.297980
in attempt 26 the loss was 2.317678, best 2.297980
in attempt 27 the loss was 2.405463, best 2.297980
in attempt 28 the loss was 2.386943, best 2.297980
in attempt 29 the loss was 2.378986, best 2.297980
in attempt 30 the loss was 2.345171, best 2.297980
in attempt 31 the loss was 2.328246, best 2.297980
in attempt 32 the loss was 2.331362, best 2.297980
in attempt 33 the loss was 2.377222, best 2.297980
in attempt 34 the loss was 2.328709, best 2.297980
in attempt 35 the loss was 2.395614, best 2.297980
in attempt 36 the loss was 2.364727, best 2.297980
in attempt 37 the loss was 2.441376, best 2.297980
in attempt 38 the loss was 2.373055, best 2.297980
in attempt 39 the loss was 2.337180, best 2.297980
in attempt 40 the loss was 2.352636, best 2.297980
in attempt 41 the loss was 2.355216, best 2.297980
in attempt 42 the loss was 2.421091, best 2.297980
in attempt 43 the loss was 2.373294, best 2.297980
```

```
in attempt 44 the loss was 2.371286, best 2.297980
in attempt 45 the loss was 2.432900, best 2.297980
in attempt 46 the loss was 2.328176, best 2.297980
in attempt 47 the loss was 2.344328, best 2.297980
in attempt 48 the loss was 2.327606, best 2.297980
in attempt 49 the loss was 2.352625, best 2.297980
in attempt 50 the loss was 2.292144, best 2.292144
in attempt 51 the loss was 2.339277, best 2.292144
in attempt 52 the loss was 2.353039, best 2.292144
in attempt 53 the loss was 2.332285, best 2.292144
in attempt 54 the loss was 2.367456, best 2.292144
in attempt 55 the loss was 2.387854, best 2.292144
in attempt 56 the loss was 2.336122, best 2.292144
in attempt 57 the loss was 2.384267, best 2.292144
in attempt 58 the loss was 2.386726, best 2.292144
in attempt 59 the loss was 2.327881, best 2.292144
in attempt 60 the loss was 2.449587, best 2.292144
in attempt 61 the loss was 2.381706, best 2.292144
in attempt 62 the loss was 2.357425, best 2.292144
in attempt 63 the loss was 2.323560, best 2.292144
in attempt 64 the loss was 2.340569, best 2.292144
in attempt 65 the loss was 2.364262, best 2.292144
in attempt 66 the loss was 2.305061, best 2.292144
in attempt 67 the loss was 2.365801, best 2.292144
in attempt 68 the loss was 2.371746, best 2.292144
in attempt 69 the loss was 2.381807, best 2.292144
in attempt 70 the loss was 2.343529, best 2.292144
in attempt 71 the loss was 2.379191, best 2.292144
in attempt 72 the loss was 2.343788, best 2.292144
in attempt 73 the loss was 2.361868, best 2.292144
in attempt 74 the loss was 2.340755, best 2.292144
in attempt 75 the loss was 2.425641, best 2.292144
in attempt 76 the loss was 2.347205, best 2.292144
in attempt 77 the loss was 2.356225, best 2.292144
in attempt 78 the loss was 2.346053, best 2.292144
in attempt 79 the loss was 2.362947, best 2.292144
in attempt 80 the loss was 2.343781, best 2.292144
in attempt 81 the loss was 2.324672, best 2.292144
in attempt 82 the loss was 2.332983, best 2.292144
in attempt 83 the loss was 2.304573, best 2.292144
in attempt 84 the loss was 2.373973, best 2.292144
in attempt 85 the loss was 2.328297, best 2.292144
in attempt 86 the loss was 2.324196, best 2.292144
in attempt 87 the loss was 2.308911, best 2.292144
in attempt 88 the loss was 2.366317, best 2.292144
in attempt 89 the loss was 2.393302, best 2.292144
in attempt 90 the loss was 2.357381, best 2.292144
in attempt 91 the loss was 2.328393, best 2.292144
```

```
in attempt 92 the loss was 2.398620, best 2.292144
in attempt 93 the loss was 2.379567, best 2.292144
in attempt 94 the loss was 2.365184, best 2.292144
in attempt 95 the loss was 2.356484, best 2.292144
in attempt 96 the loss was 2.365713, best 2.292144
in attempt 97 the loss was 2.323896, best 2.292144
in attempt 98 the loss was 2.358517, best 2.292144
in attempt 99 the loss was 2.381906, best 2.292144
```

```
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 0.112041
Accuracy on test set 0.096000
```

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

4.2 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.

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

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: -0.460559 analytic: -0.460560, relative error: 1.342489e-07
numerical: 0.533089 analytic: 0.533089, relative error: 8.297426e-08
numerical: 0.618184 analytic: 0.618184, relative error: 5.844531e-08
numerical: -0.871183 analytic: -0.871183, relative error: 2.138603e-08
numerical: -3.339929 analytic: -3.339929, relative error: 2.364019e-09
numerical: -2.656878 analytic: -2.656878, relative error: 5.484551e-09
numerical: -2.699695 analytic: -2.699695, relative error: 2.859387e-09
numerical: 1.549292 analytic: 1.549292, relative error: 4.745746e-08
numerical: -1.202114 analytic: -1.202114, relative error: 3.582565e-08
numerical: -0.558701 analytic: -0.558701, relative error: 2.694464e-08
numerical: 0.638163 analytic: 0.638163, relative error: 1.175406e-08
numerical: 0.727814 analytic: 0.727814, relative error: 5.651880e-08
numerical: 0.093812 analytic: 0.093812, relative error: 4.120904e-07
numerical: -1.341635 analytic: -1.341636, relative error: 3.845743e-08
numerical: 2.869986 analytic: 2.869986, relative error: 3.577724e-09
numerical: 0.980015 analytic: 0.980015, relative error: 8.972693e-08
numerical: -1.877714 analytic: -1.877714, relative error: 3.287474e-08
numerical: 0.611608 analytic: 0.611608, relative error: 1.981646e-08
numerical: -1.296341 analytic: -1.296342, relative error: 6.241699e-09
numerical: 2.616105 analytic: 2.616105, relative error: 2.295297e-08
```

In [27]:
```
# Now that we have a naive implementation of the softmax loss function and its gradie
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should
# 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: 2.381906e+00 computed in 0.090384s
vectorized loss: 2.381906e+00 computed in 0.004000s
```

```
Loss difference: 0.000000
Gradient difference: 0.000000
```

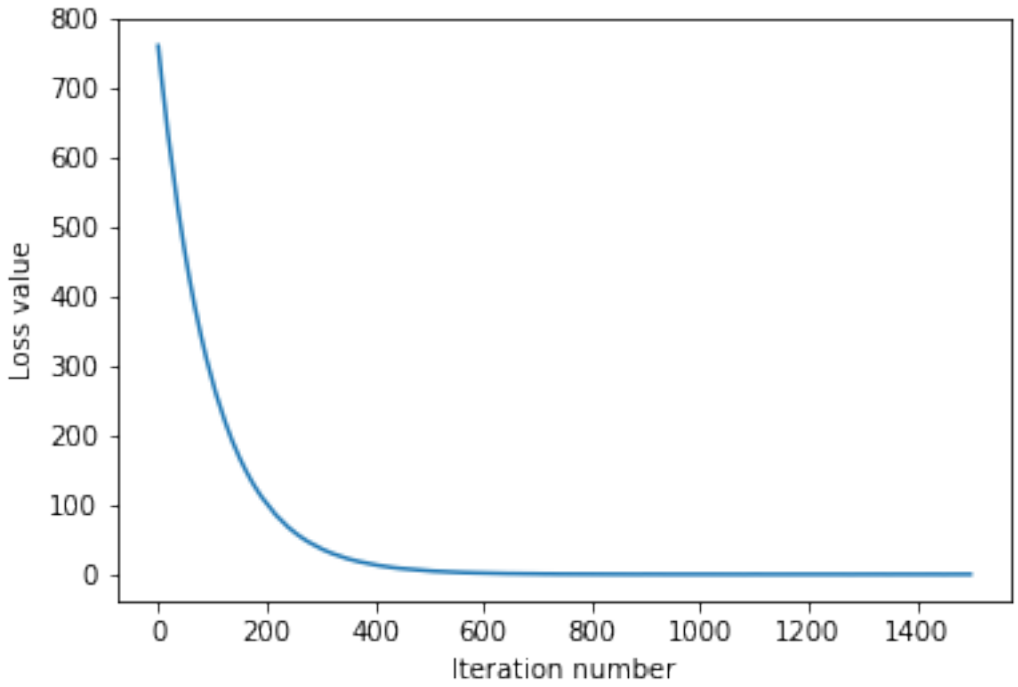
```
In [28]: from classifiers.linear_classifier import *
         # from classifiers import linear_classifier

         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 761.190667
iteration 100 / 1500: loss 279.397598
iteration 200 / 1500: loss 103.451688
iteration 300 / 1500: loss 39.276965
iteration 400 / 1500: loss 15.631933
iteration 500 / 1500: loss 7.081458
iteration 600 / 1500: loss 3.902425
iteration 700 / 1500: loss 2.795798
iteration 800 / 1500: loss 2.324842
iteration 900 / 1500: loss 2.144973
iteration 1000 / 1500: loss 2.176660
iteration 1100 / 1500: loss 2.071147
iteration 1200 / 1500: loss 2.063922
iteration 1300 / 1500: loss 2.062663
iteration 1400 / 1500: loss 2.120899
That took 2.846217s
```

```
In [43]: # 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: 0.324184
validation accuracy: 0.340000
```

```
In [44]: # 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 [45]:
```
# 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: 34.40
```

In [46]:
```
# 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 over 35% on the validation set.

import copy
best_val_acc = -1
best_softmax = None        # You may need to use copy.deepcopy(object)
learning_rates = [1e-4,1e-5, 1e-6,5e-7]
regularization_strengths = [1e-4, 1e-3, 1e-2, 1e-1, 1e0]
results = {}
################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# Save the best trained softmax classifer in best_softmax.                     #
################################################################################
```

```
        for lr in learning_rates:
            for rs in regularization_strengths:
                classifier = Softmax()
                classifier.train(X_train, y_train, learning_rate=lr, reg=rs,
                                 num_iters=1500, verbose=True)
                y_val_pred = classifier.predict(X_val)
                val_accuracy = np.mean(y_val == y_val_pred)
                results["lr:{}, rs:{}".format(lr, rs)] = val_accuracy
                if val_accuracy > best_val_acc:
                    best_val_acc = val_accuracy
                    best_softmax = copy.deepcopy(classifier)

        ################################################################################
        #                              END OF YOUR CODE                                #
        ################################################################################
        print('best cross-validation accuracy: %.2f' % best_val_acc)
```

```
iteration 0 / 1500: loss 5.751922
iteration 100 / 1500: loss 20.308866
iteration 200 / 1500: loss 27.777122
iteration 300 / 1500: loss 17.158892
iteration 400 / 1500: loss 27.620605
iteration 500 / 1500: loss 32.398785
iteration 600 / 1500: loss 28.731222
iteration 700 / 1500: loss 25.954695
iteration 800 / 1500: loss 35.455409
iteration 900 / 1500: loss 24.123352
iteration 1000 / 1500: loss 18.790200
iteration 1100 / 1500: loss 24.149677
iteration 1200 / 1500: loss 30.252764
iteration 1300 / 1500: loss 28.683276
iteration 1400 / 1500: loss 26.455454
iteration 0 / 1500: loss 5.630516
iteration 100 / 1500: loss 25.975355
iteration 200 / 1500: loss 28.352620
iteration 300 / 1500: loss 26.875154
iteration 400 / 1500: loss 25.974025
iteration 500 / 1500: loss 35.075275
iteration 600 / 1500: loss 29.916622
iteration 700 / 1500: loss 20.629986
iteration 800 / 1500: loss 25.606120
iteration 900 / 1500: loss 16.650352
iteration 1000 / 1500: loss 22.600458
iteration 1100 / 1500: loss 36.017443
iteration 1200 / 1500: loss 21.276959
iteration 1300 / 1500: loss 24.695625
iteration 1400 / 1500: loss 21.887663
iteration 0 / 1500: loss 5.213413
```

```
iteration 100 / 1500: loss 28.616522
iteration 200 / 1500: loss 17.210898
iteration 300 / 1500: loss 21.962965
iteration 400 / 1500: loss 24.100798
iteration 500 / 1500: loss 25.256354
iteration 600 / 1500: loss 44.245046
iteration 700 / 1500: loss 27.572773
iteration 800 / 1500: loss 24.362869
iteration 900 / 1500: loss 23.508256
iteration 1000 / 1500: loss 27.176197
iteration 1100 / 1500: loss 22.886922
iteration 1200 / 1500: loss 26.916559
iteration 1300 / 1500: loss 35.253493
iteration 1400 / 1500: loss 25.696459
iteration 0 / 1500: loss 6.371160
iteration 100 / 1500: loss 23.700298
iteration 200 / 1500: loss 35.603418
iteration 300 / 1500: loss 29.846247
iteration 400 / 1500: loss 25.688940
iteration 500 / 1500: loss 31.043448
iteration 600 / 1500: loss 23.094095
iteration 700 / 1500: loss 38.427383
iteration 800 / 1500: loss 46.373576
iteration 900 / 1500: loss 29.840566
iteration 1000 / 1500: loss 22.904523
iteration 1100 / 1500: loss 29.780069
iteration 1200 / 1500: loss 26.448280
iteration 1300 / 1500: loss 19.508651
iteration 1400 / 1500: loss 27.508282
iteration 0 / 1500: loss 5.893152
iteration 100 / 1500: loss 29.217130
iteration 200 / 1500: loss 27.465540
iteration 300 / 1500: loss 26.051559
iteration 400 / 1500: loss 37.655238
iteration 500 / 1500: loss 23.158917
iteration 600 / 1500: loss 39.073270
iteration 700 / 1500: loss 26.690055
iteration 800 / 1500: loss 30.435345
iteration 900 / 1500: loss 28.360215
iteration 1000 / 1500: loss 32.565640
iteration 1100 / 1500: loss 22.560941
iteration 1200 / 1500: loss 34.281252
iteration 1300 / 1500: loss 36.769840
iteration 1400 / 1500: loss 34.965905
iteration 0 / 1500: loss 5.292228
iteration 100 / 1500: loss 2.774588
iteration 200 / 1500: loss 3.622822
iteration 300 / 1500: loss 2.879779
```

```
iteration 400 / 1500: loss 2.973163
iteration 500 / 1500: loss 2.816671
iteration 600 / 1500: loss 3.690804
iteration 700 / 1500: loss 2.351930
iteration 800 / 1500: loss 3.091596
iteration 900 / 1500: loss 2.823903
iteration 1000 / 1500: loss 3.053613
iteration 1100 / 1500: loss 2.798275
iteration 1200 / 1500: loss 2.255519
iteration 1300 / 1500: loss 3.344810
iteration 1400 / 1500: loss 3.658164
iteration 0 / 1500: loss 4.987544
iteration 100 / 1500: loss 2.685975
iteration 200 / 1500: loss 2.369584
iteration 300 / 1500: loss 2.257011
iteration 400 / 1500: loss 3.137440
iteration 500 / 1500: loss 2.884803
iteration 600 / 1500: loss 3.139472
iteration 700 / 1500: loss 2.477705
iteration 800 / 1500: loss 2.321031
iteration 900 / 1500: loss 3.255086
iteration 1000 / 1500: loss 2.735100
iteration 1100 / 1500: loss 3.632789
iteration 1200 / 1500: loss 2.755755
iteration 1300 / 1500: loss 2.172191
iteration 1400 / 1500: loss 3.259874
iteration 0 / 1500: loss 4.987676
iteration 100 / 1500: loss 2.786205
iteration 200 / 1500: loss 2.829950
iteration 300 / 1500: loss 3.300553
iteration 400 / 1500: loss 2.644963
iteration 500 / 1500: loss 2.522639
iteration 600 / 1500: loss 3.143608
iteration 700 / 1500: loss 2.229267
iteration 800 / 1500: loss 3.248948
iteration 900 / 1500: loss 2.400234
iteration 1000 / 1500: loss 3.646136
iteration 1100 / 1500: loss 2.232422
iteration 1200 / 1500: loss 2.520767
iteration 1300 / 1500: loss 2.845810
iteration 1400 / 1500: loss 3.060858
iteration 0 / 1500: loss 5.979042
iteration 100 / 1500: loss 3.681059
iteration 200 / 1500: loss 3.065597
iteration 300 / 1500: loss 2.916971
iteration 400 / 1500: loss 3.015524
iteration 500 / 1500: loss 3.477493
iteration 600 / 1500: loss 2.553981
```

```
iteration 700 / 1500: loss 2.257275
iteration 800 / 1500: loss 2.321200
iteration 900 / 1500: loss 3.174933
iteration 1000 / 1500: loss 1.973882
iteration 1100 / 1500: loss 3.288896
iteration 1200 / 1500: loss 2.881387
iteration 1300 / 1500: loss 2.888068
iteration 1400 / 1500: loss 2.628099
iteration 0 / 1500: loss 5.803989
iteration 100 / 1500: loss 2.683230
iteration 200 / 1500: loss 2.971850
iteration 300 / 1500: loss 2.568575
iteration 400 / 1500: loss 3.033233
iteration 500 / 1500: loss 2.030806
iteration 600 / 1500: loss 2.018422
iteration 700 / 1500: loss 2.678212
iteration 800 / 1500: loss 2.277390
iteration 900 / 1500: loss 3.124556
iteration 1000 / 1500: loss 3.168057
iteration 1100 / 1500: loss 2.566793
iteration 1200 / 1500: loss 2.204453
iteration 1300 / 1500: loss 3.541555
iteration 1400 / 1500: loss 3.942469
iteration 0 / 1500: loss 5.078493
iteration 100 / 1500: loss 2.694090
iteration 200 / 1500: loss 2.706521
iteration 300 / 1500: loss 2.344603
iteration 400 / 1500: loss 2.386709
iteration 500 / 1500: loss 2.189337
iteration 600 / 1500: loss 2.294426
iteration 700 / 1500: loss 1.834644
iteration 800 / 1500: loss 2.064321
iteration 900 / 1500: loss 2.112666
iteration 1000 / 1500: loss 2.076632
iteration 1100 / 1500: loss 1.943344
iteration 1200 / 1500: loss 2.095332
iteration 1300 / 1500: loss 1.923915
iteration 1400 / 1500: loss 1.855377
iteration 0 / 1500: loss 5.495138
iteration 100 / 1500: loss 2.835382
iteration 200 / 1500: loss 2.576820
iteration 300 / 1500: loss 2.319076
iteration 400 / 1500: loss 2.256839
iteration 500 / 1500: loss 2.109795
iteration 600 / 1500: loss 2.354172
iteration 700 / 1500: loss 2.103093
iteration 800 / 1500: loss 2.031606
iteration 900 / 1500: loss 2.215760
```

```
iteration 1000 / 1500: loss 1.838812
iteration 1100 / 1500: loss 2.084452
iteration 1200 / 1500: loss 1.982859
iteration 1300 / 1500: loss 1.890133
iteration 1400 / 1500: loss 1.750057
iteration 0 / 1500: loss 5.663463
iteration 100 / 1500: loss 2.683091
iteration 200 / 1500: loss 2.249305
iteration 300 / 1500: loss 2.484211
iteration 400 / 1500: loss 2.400597
iteration 500 / 1500: loss 2.241984
iteration 600 / 1500: loss 2.077732
iteration 700 / 1500: loss 2.234964
iteration 800 / 1500: loss 2.029090
iteration 900 / 1500: loss 2.008646
iteration 1000 / 1500: loss 2.067791
iteration 1100 / 1500: loss 1.799560
iteration 1200 / 1500: loss 2.096422
iteration 1300 / 1500: loss 1.995200
iteration 1400 / 1500: loss 1.859014
iteration 0 / 1500: loss 5.416215
iteration 100 / 1500: loss 3.046163
iteration 200 / 1500: loss 2.713265
iteration 300 / 1500: loss 2.296213
iteration 400 / 1500: loss 2.380322
iteration 500 / 1500: loss 2.046599
iteration 600 / 1500: loss 2.136974
iteration 700 / 1500: loss 1.913620
iteration 800 / 1500: loss 1.921129
iteration 900 / 1500: loss 2.064046
iteration 1000 / 1500: loss 1.882597
iteration 1100 / 1500: loss 1.974180
iteration 1200 / 1500: loss 1.970732
iteration 1300 / 1500: loss 1.849081
iteration 1400 / 1500: loss 2.005387
iteration 0 / 1500: loss 6.463876
iteration 100 / 1500: loss 2.872776
iteration 200 / 1500: loss 2.623432
iteration 300 / 1500: loss 2.562303
iteration 400 / 1500: loss 2.380389
iteration 500 / 1500: loss 2.186694
iteration 600 / 1500: loss 2.140761
iteration 700 / 1500: loss 2.154943
iteration 800 / 1500: loss 2.116850
iteration 900 / 1500: loss 2.090924
iteration 1000 / 1500: loss 2.204529
iteration 1100 / 1500: loss 2.145447
iteration 1200 / 1500: loss 1.898620
```

```
iteration 1300 / 1500: loss 1.820990
iteration 1400 / 1500: loss 1.920955
iteration 0 / 1500: loss 5.463239
iteration 100 / 1500: loss 3.056638
iteration 200 / 1500: loss 3.054546
iteration 300 / 1500: loss 2.724621
iteration 400 / 1500: loss 2.795981
iteration 500 / 1500: loss 2.616969
iteration 600 / 1500: loss 2.339060
iteration 700 / 1500: loss 2.383793
iteration 800 / 1500: loss 2.321111
iteration 900 / 1500: loss 2.292936
iteration 1000 / 1500: loss 1.935754
iteration 1100 / 1500: loss 2.225202
iteration 1200 / 1500: loss 2.056244
iteration 1300 / 1500: loss 2.265855
iteration 1400 / 1500: loss 2.269400
iteration 0 / 1500: loss 5.480894
iteration 100 / 1500: loss 3.110675
iteration 200 / 1500: loss 2.854882
iteration 300 / 1500: loss 2.525343
iteration 400 / 1500: loss 2.466578
iteration 500 / 1500: loss 2.483129
iteration 600 / 1500: loss 2.437654
iteration 700 / 1500: loss 2.184771
iteration 800 / 1500: loss 2.322615
iteration 900 / 1500: loss 2.143513
iteration 1000 / 1500: loss 2.391282
iteration 1100 / 1500: loss 2.312991
iteration 1200 / 1500: loss 2.215274
iteration 1300 / 1500: loss 2.072154
iteration 1400 / 1500: loss 2.333828
iteration 0 / 1500: loss 5.660768
iteration 100 / 1500: loss 2.956362
iteration 200 / 1500: loss 2.833432
iteration 300 / 1500: loss 2.414224
iteration 400 / 1500: loss 2.429864
iteration 500 / 1500: loss 2.421793
iteration 600 / 1500: loss 2.286437
iteration 700 / 1500: loss 2.215906
iteration 800 / 1500: loss 2.078634
iteration 900 / 1500: loss 2.270486
iteration 1000 / 1500: loss 2.055732
iteration 1100 / 1500: loss 2.070906
iteration 1200 / 1500: loss 2.253933
iteration 1300 / 1500: loss 2.157669
iteration 1400 / 1500: loss 2.072989
iteration 0 / 1500: loss 6.509859
```

```
iteration 100 / 1500: loss 3.036556
iteration 200 / 1500: loss 2.817710
iteration 300 / 1500: loss 2.564412
iteration 400 / 1500: loss 2.418813
iteration 500 / 1500: loss 2.125927
iteration 600 / 1500: loss 2.590955
iteration 700 / 1500: loss 2.366498
iteration 800 / 1500: loss 2.250271
iteration 900 / 1500: loss 2.235108
iteration 1000 / 1500: loss 2.215379
iteration 1100 / 1500: loss 2.193092
iteration 1200 / 1500: loss 2.084933
iteration 1300 / 1500: loss 2.134696
iteration 1400 / 1500: loss 1.964596
iteration 0 / 1500: loss 5.567179
iteration 100 / 1500: loss 2.856622
iteration 200 / 1500: loss 2.646603
iteration 300 / 1500: loss 2.426377
iteration 400 / 1500: loss 2.491067
iteration 500 / 1500: loss 2.436469
iteration 600 / 1500: loss 2.314062
iteration 700 / 1500: loss 2.322637
iteration 800 / 1500: loss 2.151515
iteration 900 / 1500: loss 2.218597
iteration 1000 / 1500: loss 2.324417
iteration 1100 / 1500: loss 2.167377
iteration 1200 / 1500: loss 2.154182
iteration 1300 / 1500: loss 2.218825
iteration 1400 / 1500: loss 1.878402
best cross-validation accuracy: 0.36
```

In [47]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.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: 34.40
```