AC 209B: Homework 5

Neural Net Basics & Feed-forward Nets

Harvard University
Spring 2018
Instructors: Pavlos Protopapas and Mark Glickman

INSTRUCTIONS

- To submit your assignment follow the instructions given in canvas. Make sure you include:
  1. This notebook file `ac209b_HW5.ipynb`.
  2. `cs209b/softmax.py`
  3. `cs209b/neural_net.py`
  4. `cs209b/optim.py`
- Restart the kernel and run the whole notebook again before you submit.
- Do not include your name(s) in the notebook if you are submitting as a group.
- If you submit individually and you have worked with someone, please include the name of your [one] partner below.

Your partner's name (if you submit separately):

Download CIFAR-10

In these exercises you will implement some basic feedforward networks with numpy. You will not use Tensorflow or Keras in the following. See the `requirements.txt` to set up your environment. This is necessary for reproducibility reasons. Once you have set-up the working environment, download the CIFAR-10 dataset:

```
cd datasets/
./get_datasets.sh
```

Note: if the script does not work for you, simply go to http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) and download the file.

Part 1: Softmax exercise (5pt)

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**

Run the following code to load libraries and CIFAR-10 database.

In [1]:
```python
import random
import numpy as np
from cs209b.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 external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:

```python
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
    SVM, 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
    # Validation set
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    # Training set
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    # Test set
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    # Dev data set: just for debugging purposes, it overlaps with the training set,
    # but has a smaller size.
    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

    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()

# Training parameters
m_train = X_train.shape[0]   # number of training examples
m_dev = X_dev.shape[0]       # number of training examples in the development set
n = X_train.shape[1]         # features dimension
c = 10                       # number of classes in the database

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

The following function converts the class labels into a one hot encoding representation. The output of the encoding is (m, c), where m corresponds to the number of examples and c to the number of categories.

In [3]:

```python
from sklearn.preprocessing import OneHotEncoder
def encode_labels(y):
    enc = OneHotEncoder()
    enc.fit(y.reshape(-1,1))
    y_enc = enc.transform(y.reshape(-1,1)).toarray()
    return y_enc

Y_train_enc = encode_labels(y_train)
Y_val_enc =  encode_labels(y_val)
Y_dev_enc = encode_labels(y_dev)
Y_test_enc = encode_labels(y_test)
```

Softmax function

Your code for this part should all be written inside **cs209b/softmax.py**.

The expression of the multiclass cross-entropy plus regularization is given by:

$$J(W,b,X,y,\text{reg}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{c} y_j^{(i)} \log\left(\frac{e^{w_j x^{(i)} + b_j}}{\sum_{r=1}^{c} e^{w_r x^{(i)} + b_r}}\right) + \frac{1}{2}\text{reg} \parallel W \parallel^2.$$

Note that we do not regularize the bias term.

Other variables of interest take the following dimensions:

- c is the total number of classes; m the number of examples or the size of the mini-batch; n is the dimension of the input.
- W is matrix of size (c, n); w_j refers to the jth row, has size $(1, n)$.
- b is a column vector, with size $(c, 1)$; b_j is the jth component.
- X is the training mini-batch, with size (m, n); $x^{(i)}$ is the ith example.
- Y has the labeling data with a one-hot encoding representation, and has size (m, c); $y^{(i)}$ is the one-hot encoding for training example i, with size $(1, c)$.

Forward pass:

1. Implement the function `softmax_loss_naive` , which returns the loss of the cross-entropy function. Do not implement the gradient propagation just yet, we will explain that part after this exercise is completed and you have verified that it works correctly. We recommend that you write this function using a for loop over the minibatch examples, and later make a vectorized version in `softmax_loss_vectorized` . However, if you want to program the vectorized version directly, feel free to have the same code in both functions.
2. Our loss should be something close to -log(0.1) (run the code below as a sanity check). Why do we expect our loss to be close to -log(0.1)? Explain briefly.

Do not forget the regularization term.

Hints: Avoid numerical problems by normalizing the exponents of the softmax function.

In [4]:

```python
from cs209b.softmax import softmax_loss_naive
from cs209b.softmax import softmax_loss_vectorized
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(c,n) * 0.0001
b = np.random.randn(c,1) * 0.0001
loss_naive = softmax_loss_naive(W, b, X_dev, Y_dev_enc, reg = 0.0)[0]
loss_vectorized = softmax_loss_vectorized(W, b, X_dev, Y_dev_enc, reg = 0.0)[0]

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

```
random loss (naive):  2.372150
random loss (vectorized):  2.372150
sanity check: 2.302585
```

Answers:

We would expect the random initialization of W and b outputs the softmax probability of $\frac{1}{10}$ for each of the 10 classes. Therefore, the expected cross entropy loss would be $-log(0.1)$.

Note: Both `softmax_loss_naive` and `softmax_loss_vectorized` are vectorized in our implementation.

Backward pass:

Now we are going to program the gradient computation for the softmax function. You will have to figure out the correct expressions and make use of the chain rule.

1. Finish the implementation of the function `softmax_loss_naive` returning correct gradients dW and db. These terms should have the following shapes:

   - dW has dimension (c, n).
   - db has dimension $(c, 1)$.

   We provide the following derivative to help with your code:

   $$\frac{\partial J}{\partial z_j^{(i)}} = a_j^{(i)} - y_j^{(i)}$$

   where $z_j^{(i)} = w_j x^{(i)} + b_j$ and $a_j^{(i)} = \frac{e^{z_j^{(i)}}}{\sum_{r=1}^{c} z_r^{(i)}}$.
2. Please, finish the implementation of `softmax_loss_naive` using loops. We will later implement the vectorized version, but feel free to do it now if that is your preference. Run the following code to check analytical gradients with numerical ones. You should get errors below 10^{-7} (except on the bias, which may be around 10^{-7}).

In [5]:

```python
np.random.seed(1)
W = np.random.randn(c,n) * 0.0001
b = np.random.randn(c,1) * 0.0001
loss, grads = softmax_loss_naive(W, b, X_dev, Y_dev_enc, 0.0)
dW, db = grads["dW"], grads["db"]

# We use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs209b.gradient_check import grad_check_sparse
f = lambda W: softmax_loss_naive(W, b, X_dev, Y_dev_enc, 0.0)[0]
print("Numerical gradient of W without regularization:")
grad_numerical = grad_check_sparse(f, W, dW, 10)

# do another gradient check with regularization
loss, grads = softmax_loss_naive(W, b, X_dev, Y_dev_enc, 5e1)
dW, db = grads["dW"], grads["db"]
f = lambda W: softmax_loss_naive(W, b, X_dev, Y_dev_enc, 5e1)[0]
print("\nNumerical gradient of W with regularization:")
grad_numerical = grad_check_sparse(f, W, dW, 10)

# Verify gradient of bias:
from cs209b.gradient_check import grad_check_sparse
f = lambda b: softmax_loss_naive(W, b, X_dev, Y_dev_enc, 0.0)[0]
print("\nNumerical gradient of bias:")
grad_numerical = grad_check_sparse(f, b, db, 10)
```

```
Numerical gradient of W without regularization:
numerical: 0.440308 analytic: 0.440308, relative error: 1.867743e-09
numerical: 1.019816 analytic: 1.019816, relative error: 1.173309e-10
numerical: -0.414289 analytic: -0.414289, relative error: 2.319320e-10
numerical: -1.618564 analytic: -1.618564, relative error: 1.076354e-10
numerical: 0.024880 analytic: 0.024880, relative error: 3.443183e-08
numerical: 0.589210 analytic: 0.589210, relative error: 1.067057e-09
numerical: -0.464590 analytic: -0.464590, relative error: 6.238478e-11
numerical: 2.185027 analytic: 2.185027, relative error: 3.243067e-10
numerical: -0.629990 analytic: -0.629990, relative error: 2.414618e-10
numerical: 2.215128 analytic: 2.215128, relative error: 7.727848e-10

Numerical gradient of W with regularization:
numerical: 0.988420 analytic: 0.988420, relative error: 2.347931e-10
numerical: -0.205535 analytic: -0.205535, relative error: 6.004220e-09
numerical: -0.140116 analytic: -0.140116, relative error: 8.694832e-09
numerical: -2.124938 analytic: -2.124938, relative error: 4.513802e-11
numerical: 4.103547 analytic: 4.103547, relative error: 1.651043e-10
numerical: -1.627325 analytic: -1.627325, relative error: 5.968656e-10
numerical: -1.088276 analytic: -1.088276, relative error: 2.905053e-10
numerical: -1.262700 analytic: -1.262700, relative error: 6.784572e-10
numerical: 0.552624 analytic: 0.552624, relative error: 1.908631e-09
numerical: 0.230894 analytic: 0.230894, relative error: 2.195412e-09

Numerical gradient of bias:
numerical: 0.011228 analytic: 0.011228, relative error: 5.689103e-08
numerical: 0.016020 analytic: 0.016020, relative error: 4.097351e-08
numerical: 0.016020 analytic: 0.016020, relative error: 4.097351e-08
numerical: -0.007204 analytic: -0.007204, relative error: 4.987295e-08
numerical: -0.005719 analytic: -0.005719, relative error: 1.583247e-07
numerical: 0.005828 analytic: 0.005828, relative error: 2.967826e-07
numerical: 0.026876 analytic: 0.026876, relative error: 2.057792e-09
numerical: 0.011228 analytic: 0.011228, relative error: 5.689103e-08
numerical: 0.016020 analytic: 0.016020, relative error: 4.097351e-08
numerical: 0.016020 analytic: 0.016020, relative error: 4.097351e-08
```

Vectorized version

1. Now that we have a naive implementation of the softmax loss function and its gradient, implement a vectorized version `softmax_loss_vectorized` in file **cs209b/softmax.py**. The two versions should compute the same results, but the vectorized version should be much faster. It is OK if you implemented the vectorized version on both functions.

In [6]:

```python
tic = time.time()
loss_naive, grads_naive = softmax_loss_naive(W, b, X_train, Y_train_enc, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs209b.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grads_vectorized = softmax_loss_vectorized(W, b, X_train, Y_train_enc, 0.000005)
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(grads_naive["dW"] - grads_vectorized["dW"], ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.308877e+00 computed in 0.498985s
vectorized loss: 2.308877e+00 computed in 0.389956s
Loss difference: 0.000000
Gradient difference: 0.000000
```

Optimization:

1. Complete the function `optimize` in **cs209b/softmax.py** that performs stochastic gradient descent using mini-batches of data randomly sampled from the training set. You should complete the following steps:
   A. Sample your mini-batch attending to the `batch_size` parameter. We suggest you use `np.random.choice` to generate indices. Sampling with replacement is faster than sampling without replacement.
   B. Determine loss and gradients.
   C. Retrieve derivatives from grads
   D. Perform stochastic gradient descent update.

You should see that the cost is reduced rapidly in 400 iterations. The expected costs should be around the same magnitude, although not exactly the same because of randomness. Continue on the evaluation part and check if you reach an accuracy of around 60% on the training data.

In [7]:
```
from cs209b.softmax import optimize
np.random.seed(1)
W = np.random.randn(c, n) * 0.0001
b = np.random.randn(c, 1) * 0.0001
params, grads, costs = optimize(W, b, X_dev, Y_dev_enc, num_iterations=400, learning_rate=1e-5, reg=0, batch_size=100, print_cost =
print("Computed costs: ", costs)
print("Expected costs: ", [2.2500092769842173, 0.6131757237029045, 0.2379333500616387, 0.12624630382907445])
```

```
Cost after iteration 0: 1.840567
Cost after iteration 100: 0.371838
Cost after iteration 200: 0.180250
Cost after iteration 300: 0.097113
Computed costs:  [1.8405671424948833, 0.3718377771255655, 0.18024994126119928, 0.09711341830148289]
Expected costs:  [2.2500092769842173, 0.6131757237029045, 0.2379333500616387, 0.12624630382907445]
```

Prediction

1. Implement the function `predict` on file **cs209b/softmax.py**. The returned value of `predict` should directly return the class for the input, i.e., an integer value from 0 to $c - 1$ ($c = 10$ here since there are 10 classes in CIFAR-10). This is not the one-hot encoding that we have been using for the training labels.

You should see a low number of predicted errors. The highest number we have seen is 16, and the lowest was zero.

In [8]:
```
from cs209b.softmax import predict, model
W = params["W"]
b = params["b"]
y_pred = predict(W, b, X_dev)
print("Number of predicted errors on dev set: ", np.sum(np.abs(y_pred-y_dev)))
```

```
Number of predicted errors on dev set:  0
```

Evaluation

1. Implement the function `model` on file **cs209b/softmax.py** that puts all the pieces together. The function should include:
   A. Initialize parameters W and b. Anything is fine, the problem is convex.
   B. Optimize the parameters.
   C. Perform predictions on the training and test sets.
   D. Return the results.

   Save the training and test predictions into `Y_prediction_train` and `Y_prediction_test`, respectively.
2. Evaluate the model performance on the dev set. Run the following line of code. You should obtain around 59% of train accuracy, and 28% on the validation set (or above).

In [9]:
```
d = model(X_dev, Y_dev_enc, X_val, Y_val_enc, num_iterations = 1000, learning_rate = 1e-6, reg=1e4, batch_size = 100, print_cost =
```

```
Cost after iteration 0: 18.436976
Cost after iteration 100: 3.561583
Cost after iteration 200: 1.962051
Cost after iteration 300: 1.661961
Cost after iteration 400: 1.441785
Cost after iteration 500: 1.603502
Cost after iteration 600: 1.611462
Cost after iteration 700: 1.568000
Cost after iteration 800: 1.554394
Cost after iteration 900: 1.516330
train accuracy: 0.64
validation accuracy: 0.323
```

Part 2: Two-Layer Neural Network (5pts)

In the next exercise you will develop a neural network with two fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset. You will:

- Implement the forward and backward pass of a two layer network, as well as the cross-entropy loss.
- Train the network using stochastic gradient descent.
- Implement dropout on the input layer and hidden layer.
- Run the implementation on the CIFAR-10 database.
- Implement SGD with momentum, RMSprop and Adam as optimization algorithms.

Restart the Kernel now, we will reload the database and libraries.

In [10]:

```python
import numpy as np
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 external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

We will implement the necessary functions to represent and train a two layer network. The network parameters are stored in the instance variable `params` where keys are string parameter names and values are numpy arrays. Below, we initialize the network parameters.

In [11]:

```python
def init_2layer_net(input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (n1, n0)
    b1: First layer biases; has shape (n1, 1)
    W2: Second layer weights; has shape (c, n0)
    b2: Second layer biases; has shape (c, 1)

    Inputs:
    - input_size: The dimension n0 of the input data.
    - hidden_size: The number of neurons n1 in the hidden layer.
    - output_size: The number of classes c.
    """
    params = {}
    params['W1'] = std * np.random.randn(hidden_size, input_size)
    params['b1'] = np.zeros((hidden_size, 1))
    params['W2'] = std * np.random.randn(output_size, hidden_size)
    params['b2'] = np.zeros((output_size, 1))
    return params
```

We initialize toy data and a toy model that we will use to develop your implementation.

In [12]:

```python
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return init_2layer_net(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

params = init_toy_model()
X, y = init_toy_data()
```

As with the softmax classifier, we compute the one-hot encoding of the labels:

In [13]:

```python
from sklearn.preprocessing import OneHotEncoder
def encode_labels(y):
    enc = OneHotEncoder()
    enc.fit(y.reshape(-1,1))
    y_enc = enc.transform(y.reshape(-1,1)).toarray()
    return y_enc

Y_enc = encode_labels(y)
```

Forward pass: compute scores

Open the file **cs209b/neural_net.py** and look at the method `loss_2layer_net`. This function is very similar to the loss functions you have written for the Softmax exercise: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

1. Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs. The following equations may be helpful:

$$
\begin{aligned}
z^{[1](i)} &= W^{[1]}x^{(i)} + b^{[1]} \\
a^{[1](i)} &= \mathrm{ReLu}(z^{[1](i)}) \\
z^{[2](i)} &= W^{[2]}a^{[1](i)} + b^{[2]} \\
a^{[2](i)} &= \mathrm{softmax}(z^{[2](i)}),
\end{aligned}
$$

where $[1], [2]$ indicate the hidden or output layer, respectively, and, (i) to the ith training example.

Have in mind the following dimensions:

- $W^{[1]}$ has shape $(n1, n0)$
- $b^{[1]}$ has shape $(n1, 1)$
- $W^{[2]}$ has shape $(c, n1)$
- $b^{[2]}$ has shape $(c, 1)$
- $X = (x^{(i)})$ has shape $(m, n0)$ (m is the number of training examples)
- Y has shape (m, c) (c is the number of classes)

Finally, we remind you the cross-entropy loss function:

$$
J(\text{params}, X, Y, \text{reg}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{c} y_j^{(i)} \log\left(\frac{z_j^{[2](i)}}{\sum_{r=1}^{c} z_r^{[2](i)}}\right) + \frac{1}{2}\text{reg} \parallel W^{[1]} \parallel^2 + \frac{1}{2}\text{reg} \parallel W^{[2]} \parallel^2 .
$$

Be careful with numerical overflow, normalize the output before computing the loss.

In [14]:

```python
from cs209b.neural_net import *
scores = loss_2layer_net(params, X, Y=None, reg=0.0)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
 [-1.07260209,  0.05083871, -0.87253915],
 [-2.02778743, -0.10832494, -1.52641362],
 [-0.74225908,  0.15259725, -0.39578548],
 [-0.38172726,  0.10835902, -0.17328274],
 [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231241522675e-08
```

Forward pass: compute loss

1. In the same function, implement the second part that computes the data and regularizaion loss.

In [15]:

```python
loss, _ = loss_2layer_net(params, X, Y=Y_enc, reg=0.05)
correct_loss = 1.071696123862817

# should be very small, we get 0.0
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
0.0
```

Backward pass

1. Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now you can debug your backward pass using a numeric gradient check:

In [16]:

```python
from cs209b.gradient_check import grad_check

# Use numeric gradient checking to check your implementation of the backward pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = loss_2layer_net(params, X, Y_enc, reg=0.05)

def f_change_param(param_name, U):
    params[param_name] = U
    return loss_2layer_net(params, X, Y_enc, reg=0.05)[0]

for param_name in params:
    f = lambda U: f_change_param(param_name, U)
    param_grad_num = grad_check(f, params[param_name], epsilon=1e-5)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads["d"+str(param_name)])))
```

```
W1 max relative error: 1.283284e-09
b1 max relative error: 2.202551e-09
W2 max relative error: 3.425472e-10
b2 max relative error: 1.839154e-10
```

Train the network

To train the network you will implement stochastic gradient descent (SGD) as well as other first order methods. Look at the function `train_2layer_net` in file **cs209b/neural_net.py** and make sure you understand it. You do not need to modify this function.

1. Implement function `sgd` in file **cs209b/optim.py** that updates network parameters $W1$, $W2$, $b1$ and $b2$ using a simple gradient descent update rule. The gradients are provided in `grads` and the parameters in `params` .
2. You also have to implement `predict` in **cs209b/neural_net.py**, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented both methods, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

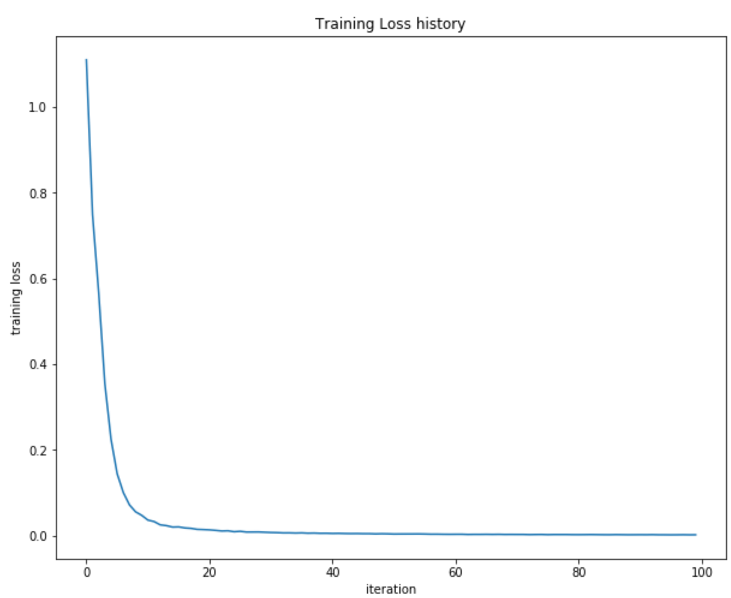
In [17]:

```python
from cs209b.neural_net import *
#from cs209b.optim import *
params = init_toy_model()
net = train_2layer_net(params, X, Y_enc, X_val=X, Y_val=Y_enc,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', net['loss_history'][-1])

# plot the loss history
plt.plot(net['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.0016458530836711452
```



Dropout

Dropout is a regularization technique for deep learning that randomly shuts down some neurons in each iteration.

The idea is that at each iteration, you train a different version of the network that uses only a subset of your neurons. The network thus become less sensitive to the activation of specific groups of neurons, because they can be shut down at any time. This encourages weights to be spread among all values.

You will now modify the forward and backward pass of `loss_2layer_net` function to incorporate dropout on the input and hidden layer, from file **cs209b/neural_net.py**. We do not apply dropout to the output layer. We will implment inverted dropout, so that the `predict` function is transparent to the whole process.

In order to implement the regularization method, you have to generate a random matrix $D^{[0]}$ and $D^{[1]}$ with 1's and 0's with the same shape as the input and hidden feature $a^{[1]}$ indicating if the neuron is active or inactive. The probability of getting an active neuron is $1 - p$, where p referst to the dropout probability. Do not forget to normalize the output of every layer that uses dropout with $1/(1-p)$.

1. Implement the forward and backward pass now.

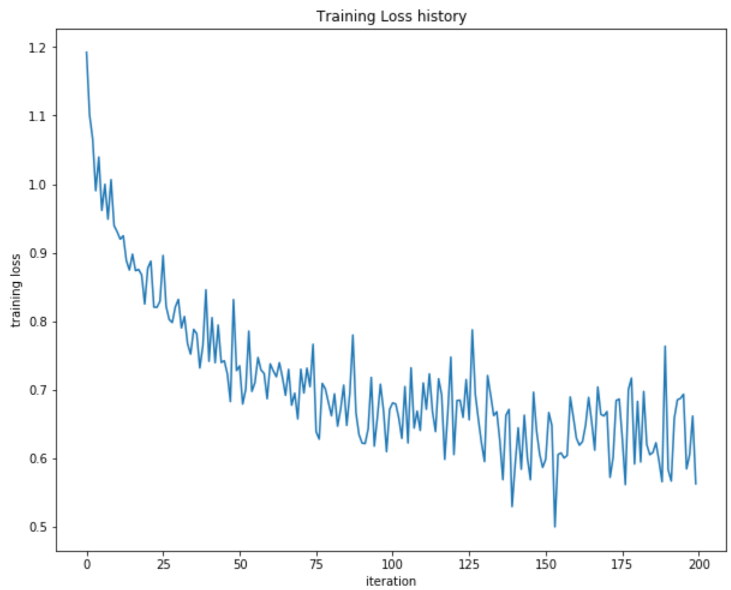
Run the following code to check for correctness. You should see a plot with noisy descent and loss value around 0.52902613.

In [18]:
```
params = init_toy_model()
net = train_2layer_net(params, X, Y_enc, X_val=X, Y_val=Y_enc,
            dropout=0.4, learning_rate=1e-2, reg=0.0,
            num_iters=200, verbose=False)

print('Final training loss: ', net['loss_history'][-1])

# plot the loss history
plt.plot(net['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.5631390138609991
```



Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

In [19]:

```
from cs209b.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, 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 = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # 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

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = 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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Perform one-hot encoding for the training, validation and test labels:

In [20]:

```
Y_train_enc = encode_labels(y_train)
Y_val_enc = encode_labels(y_val)
Y_test_enc = encode_labels(y_test)
```

Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [21]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
params = init_2layer_net(input_size, hidden_size, num_classes)
# Train the network
net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            dropout=0.0, reg=0.25, verbose=True)

# Predict on the validation set
val_acc = np.mean(predict(net["params"],X_val) == y_val)
print('Validation accuracy: ', val_acc)
```

```
iter 0 / 1000: loss 2.302758
iter 100 / 1000: loss 2.302285
iter 200 / 1000: loss 2.299410
iter 300 / 1000: loss 2.269269
iter 400 / 1000: loss 2.181931
iter 500 / 1000: loss 2.124368
iter 600 / 1000: loss 2.054913
iter 700 / 1000: loss 2.072344
iter 800 / 1000: loss 1.977126
iter 900 / 1000: loss 1.921021
Validation accuracy:  0.293
```

Debug the training

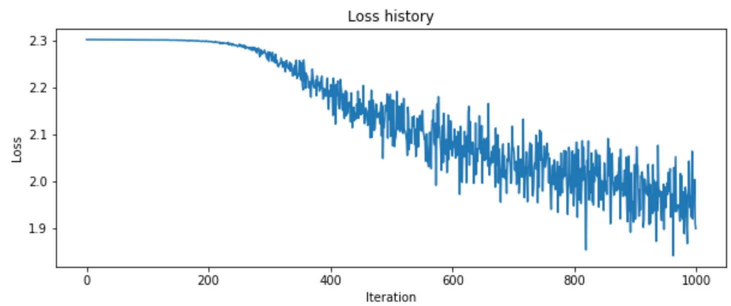
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

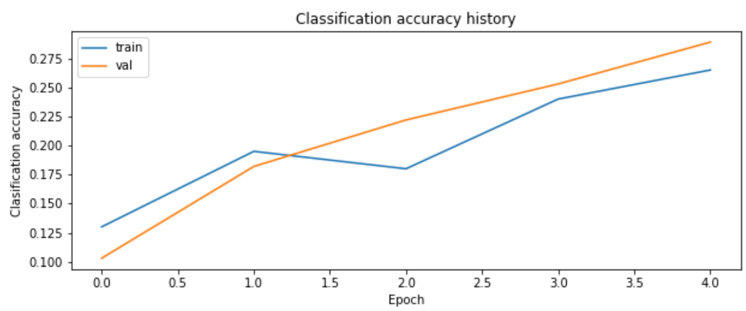
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

In [22]:

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(net['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

plt.subplot(2, 1, 2)
plt.plot(net['train_acc_history'], label='train')
plt.plot(net['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```





Tune your hyperparameters

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks. You should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, regularization strength and dropout probability. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

1. Find good hyperparameters that achieve a classification accuracy of greater than 48% on the validation set. You may use the following code to search for these parameters. For grading purposes, leave only the best hyperparameters you found.

In [23]:

```
best_net = None # store the best model into this

#################################################################################
# Tune hyperparameters using the validation set.                                #
# Store your best trained model leaving only your best combination of           #
# hyperparameters.                                                              #
#                                                                               #
# Use this code to find good hyperparamters.                                    #
#################################################################################

best_val = -1
results = {}

np.random.seed(0)

batch_sizes = [200]      # KEEP ONLY THE BEST VALUE
learning_rates = [1e-3]  # KEEP ONLY THE BEST VALUE
regs = [0.5]              # KEEP ONLY THE BEST VALUE
dropouts = [0.0]         # KEEP ONLY THE BEST VALUE

grid_search=[(s,x,y,z) for s in dropouts for x in batch_sizes for y in learning_rates for z in regs]

for dropout, batch_size, learning_rate, reg in grid_search:

    params = init_2layer_net(input_size, hidden_size, num_classes)

    # Train the network
    net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
                num_iters=2000, batch_size=batch_size,
                learning_rate=learning_rate, learning_rate_decay=0.95,
                dropout=dropout, reg=reg, verbose=True)

    # Predict on the validation set and compute accuracy
    val_acc = (predict(net["params"], X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)

    results[(dropout,batch_size,learning_rate,reg)]=val_acc

    if val_acc>best_val:
        best_val=val_acc
        best_net=net
```

```
iter 0 / 2000: loss 2.302970
iter 100 / 2000: loss 1.965017
iter 200 / 2000: loss 1.764674
iter 300 / 2000: loss 1.688132
iter 400 / 2000: loss 1.714392
iter 500 / 2000: loss 1.534171
iter 600 / 2000: loss 1.537983
iter 700 / 2000: loss 1.579538
iter 800 / 2000: loss 1.525457
iter 900 / 2000: loss 1.513263
iter 1000 / 2000: loss 1.590375
iter 1100 / 2000: loss 1.583635
iter 1200 / 2000: loss 1.467874
iter 1300 / 2000: loss 1.387518
iter 1400 / 2000: loss 1.541379
iter 1500 / 2000: loss 1.458956
iter 1600 / 2000: loss 1.601548
iter 1700 / 2000: loss 1.454189
iter 1800 / 2000: loss 1.430697
iter 1900 / 2000: loss 1.432853
Validation accuracy:  0.509
```

Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

In [24]:

```
params = best_net["params"]
test_acc = (predict(params, X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.485
```

Apply the same architecture but train it for **more (4000) epochs**:

In [25]:

```
best_net = None # store the best model into this

#################################################################################
# Tune hyperparameters using the validation set.                               #
# Store your best trained model leaving only your best combination of          #
# hyperparameters.                                                              #
#                                                                               #
# Use this code to find good hyperparamters.                                    #
#################################################################################

best_val = -1
results = {}

np.random.seed(0)

batch_sizes = [200]      # KEEP ONLY THE BEST VALUE
learning_rates = [1e-3]  # KEEP ONLY THE BEST VALUE
regs = [0.5]              # KEEP ONLY THE BEST VALUE
dropouts = [0.0]         # KEEP ONLY THE BEST VALUE

grid_search=[(s,x,y,z) for s in dropouts for x in batch_sizes for y in learning_rates for z in regs]

for dropout, batch_size, learning_rate, reg in grid_search:

    params = init_2layer_net(input_size, hidden_size, num_classes)

    # Train the network
    net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
                num_iters=4000, batch_size=batch_size,
                learning_rate=learning_rate, learning_rate_decay=0.95,
                dropout=dropout, reg=reg, verbose=True)

    # Predict on the validation set and compute accuracy
    val_acc = (predict(net["params"], X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)

    results[(dropout,batch_size,learning_rate,reg)]=val_acc

    if val_acc>best_val:
        best_val=val_acc
        best_net=net
```

```
iter 0 / 4000: loss 2.302970
iter 100 / 4000: loss 1.965017
iter 200 / 4000: loss 1.764674
iter 300 / 4000: loss 1.688132
iter 400 / 4000: loss 1.714392
iter 500 / 4000: loss 1.534171
iter 600 / 4000: loss 1.537983
iter 700 / 4000: loss 1.579538
iter 800 / 4000: loss 1.525457
iter 900 / 4000: loss 1.513263
iter 1000 / 4000: loss 1.590375
iter 1100 / 4000: loss 1.583635
iter 1200 / 4000: loss 1.467874
iter 1300 / 4000: loss 1.387518
iter 1400 / 4000: loss 1.541379
iter 1500 / 4000: loss 1.458956
iter 1600 / 4000: loss 1.601548
iter 1700 / 4000: loss 1.454189
iter 1800 / 4000: loss 1.430697
iter 1900 / 4000: loss 1.432853
iter 2000 / 4000: loss 1.403654
iter 2100 / 4000: loss 1.474237
iter 2200 / 4000: loss 1.436211
iter 2300 / 4000: loss 1.303527
iter 2400 / 4000: loss 1.462674
iter 2500 / 4000: loss 1.507066
iter 2600 / 4000: loss 1.486668
iter 2700 / 4000: loss 1.539378
iter 2800 / 4000: loss 1.433451
iter 2900 / 4000: loss 1.458442
iter 3000 / 4000: loss 1.275907
iter 3100 / 4000: loss 1.395572
iter 3200 / 4000: loss 1.289468
iter 3300 / 4000: loss 1.302251
iter 3400 / 4000: loss 1.306178
iter 3500 / 4000: loss 1.520798
iter 3600 / 4000: loss 1.434705
iter 3700 / 4000: loss 1.498881
iter 3800 / 4000: loss 1.319477
iter 3900 / 4000: loss 1.386265
Validation accuracy:  0.524
```

In [26]:

```
params = best_net["params"]
test_acc = (predict(params, X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.508
```

OPTIONAL: Optimization algorithms (3 extra points)

You have used stochastic gradient descent to train the network, but now we are going to implement sgd with momentum, RMSprop and Adam as optimization improvements to our training.

SGD with momentum

SGD has trouble minimizing the cost on areas where the surface curves more steeply in one dimension than in another. In these situations, SGD oscillates across the slopes and only making slow progress along the bottom towards a local optimum.

SGD with Momentum helps accelerate SGD in the relevant direction and dampens oscillations. It adds a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_w J$$
$$w_t = w_{t-1} - \alpha v_t,$$

where w is the parameter of interest, α refers to the learning rate, and t iteration number. The previous update rule is performed on all parameters of the neural network, i.e., $W^{[1]}$, $W^{[2]}$, $b^{[1]}$ and $b^{[2]}$. The velocity v_t is different for each parameter.

1. Implement function `sgd_momentum` in **cs209b/optim.py** and run the following code to minimize the cost function. You should get an accuracy of around 0.47 or above in 2000 iterations with a learning rate of 1e-3.

In [27]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
params = init_2layer_net(input_size, hidden_size, num_classes)
# Train the network
net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
            num_iters=2000, batch_size=100, optimizer="momentum",
            learning_rate=1e-3, beta = 0.9, learning_rate_decay=1,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = np.mean(predict(net["params"],X_val) == y_val)
print('Validation accuracy: ', val_acc)
```

```
iter 0 / 2000: loss 2.302769
iter 100 / 2000: loss 2.116357
iter 200 / 2000: loss 1.888966
iter 300 / 2000: loss 1.997414
iter 400 / 2000: loss 1.524654
iter 500 / 2000: loss 1.864854
iter 600 / 2000: loss 1.538710
iter 700 / 2000: loss 1.445322
iter 800 / 2000: loss 1.534040
iter 900 / 2000: loss 1.486341
iter 1000 / 2000: loss 1.607137
iter 1100 / 2000: loss 1.492290
iter 1200 / 2000: loss 1.445142
iter 1300 / 2000: loss 1.270807
iter 1400 / 2000: loss 1.572310
iter 1500 / 2000: loss 1.325115
iter 1600 / 2000: loss 1.524253
iter 1700 / 2000: loss 1.515654
iter 1800 / 2000: loss 1.521185
iter 1900 / 2000: loss 1.414452
Validation accuracy:  0.479
```

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton to resolve the diminishing learning rates of Adagrad and other descent techniques. It tracks a second moment of the gradient to control the learning rate. Its equations are as follows:

$$s_t = \beta_2 s_{t-1} + (1-\beta_2)(\nabla_w J)^2$$
$$w_t = w_{t-1} + \alpha \frac{1}{\sqrt{s_t} + \epsilon}\nabla_w J,$$

for every parameter to update.

1. Implement function `rmsprop` on file **cs209b/optim.py**. We get an accuracy of 0.47.

In [30]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
params = init_2layer_net(input_size, hidden_size, num_classes)
# Train the network
net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
            num_iters=2000, batch_size=200, optimizer="rmsprop",
            learning_rate=1e-4, beta2=0.999, epsilon=1e-8,
            learning_rate_decay=1, reg=0.25, verbose=True)

# Predict on the validation set
val_acc = np.mean(predict(net["params"],X_val) == y_val)
print('Validation accuracy: ', val_acc)
```

```
iter 0 / 2000: loss 2.302771
iter 100 / 2000: loss 1.882606
iter 200 / 2000: loss 1.683040
iter 300 / 2000: loss 1.639141
iter 400 / 2000: loss 1.763404
iter 500 / 2000: loss 1.643682
iter 600 / 2000: loss 1.587571
iter 700 / 2000: loss 1.565562
iter 800 / 2000: loss 1.670900
iter 900 / 2000: loss 1.737818
iter 1000 / 2000: loss 1.627008
iter 1100 / 2000: loss 1.451146
iter 1200 / 2000: loss 1.627374
iter 1300 / 2000: loss 1.507571
iter 1400 / 2000: loss 1.612409
iter 1500 / 2000: loss 1.563721
iter 1600 / 2000: loss 1.609733
iter 1700 / 2000: loss 1.509003
iter 1800 / 2000: loss 1.520945
iter 1900 / 2000: loss 1.515463
Validation accuracy:  0.486
```

Adam

Adaptive Moment Estimation ([Adam (https://arxiv.org/abs/1412.6980)](https://arxiv.org/abs/1412.6980)) is first order method that that computes adaptive learning rates for each parameter.

Adam keeps an exponentially decaying average of past gradients v_t, similar to momentum:

$$v_t = \beta_1 v_{t-1} + (1-\beta_1)\nabla_w J$$
$$s_t = \beta_2 s_{t-1} + (1-\beta_2)(\nabla_w J)^2.$$

v_t and s_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively. As v_t and s_t are initialized, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They control these underestimates with bias correction:

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$
$$\hat{s}_t = \frac{s_t}{1-\beta_2^t},$$

where $\hat{v}_t$ and $\hat{s}_t$ stand for the corrected versions.

Finally, the authors update the parameters with the following rule:

$$w_t = w_{t-1} - \alpha \frac{\hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}.$$

1. Implement function `adam` in **cs209b/optim.py** and run the following code to minimize the cost function. We get an accuracy of 0.48 in 2000 iterations with a learning rate of 1e-4.

In [32]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
params = init_2layer_net(input_size, hidden_size, num_classes)
# Train the network
net = train_2layer_net(params, X_train, Y_train_enc, X_val, Y_val_enc,
            num_iters=2000, batch_size=200, optimizer="adam",
            learning_rate=1e-4, beta1=0.9, beta2=0.999, epsilon=1e-8,
            learning_rate_decay=1, reg=0.25, verbose=True)

# Predict on the validation set
val_acc = np.mean(predict(net["params"],X_val) == y_val)
print('Validation accuracy: ', val_acc)
```

```
iter 0 / 2000: loss 2.302773
iter 100 / 2000: loss 1.799709
iter 200 / 2000: loss 1.654577
iter 300 / 2000: loss 1.631738
iter 400 / 2000: loss 1.598364
iter 500 / 2000: loss 1.673782
iter 600 / 2000: loss 1.560236
iter 700 / 2000: loss 1.501966
iter 800 / 2000: loss 1.561815
iter 900 / 2000: loss 1.531484
iter 1000 / 2000: loss 1.608274
iter 1100 / 2000: loss 1.669288
iter 1200 / 2000: loss 1.470763
iter 1300 / 2000: loss 1.463567
iter 1400 / 2000: loss 1.510683
iter 1500 / 2000: loss 1.429695
iter 1600 / 2000: loss 1.535505
iter 1700 / 2000: loss 1.413637
iter 1800 / 2000: loss 1.576382
iter 1900 / 2000: loss 1.428817
Validation accuracy:  0.508
```


Acknowledgments for the CS209B part:

Some of the code snipets and documentation was borrowed from the cs231n Stanford course. In particular, we reused all code to download CIFAR-10 and data_utils.py. Our problem layout is also based on their original structure, inspired as well on deeplearning.ai course layout.

In [ ]:

```

```