# Two-layer Neural Network Workbook for CS145 Homework 3

**PRINT YOUR NAME AND UID HERE!**

NAME: [Ye, Yusong] UID: [004757800]

Please follow the notebook linearly to implement a two layer neural network.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with training a two layer neural network.

Import libraries and define relative error function, which is used to check results later.

In [1]:
```
# import random
import numpy as np
from cs145.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0)
%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))))
```

## Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass.

In [2]:
```
from lib.neural_net import TwoLayerNet
```

In [3]:
```
# 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 TwoLayerNet(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

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

## Compute forward pass scores

In [4]:
```
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
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:
3.381231210991542e-08
```

## Forward pass loss

The total loss includes data loss (MSE) and regularization loss, which is,

$$L = L_{data} + L_{reg} = \frac{1}{2N}\sum_{i=1}^{N}\left(\boldsymbol{y}_{\text{pred}} - \boldsymbol{y}_{\text{target}}\right)^2 + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

More specifically in multi-class situation, if the output of neural nets from one sample is $y_{\text{pred}} = (0.1, 0.1, 0.8)$ and $y_{\text{target}} = (0, 0, 1)$ from the given label, then the MSE error will be $Error = (0.1-0)^2 + (0.1-0)^2 + (0.8-1)^2 = 0.06$

Implement data loss and regularization loss. In the MSE function, you also need to return the gradients which need to be passed backward. This is similar to batch gradient in linear regression. Test your implementation of loss functions. The Difference should be less than 1e-12.

In [5]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss_MSE = 3.775701133135245 # check this number

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss_MSE)))
```

```
Difference between your loss and correct loss:
1.8783678567646807
```

## Backward pass (You do not need to implemented this part)

We have already implemented the backwards pass of the neural network for you. Run the block of code to check your gradients with the gradient check utilities provided. The results should be automatically correct (tiny relative error).

If there is a gradient error larger than 1e-8, the training for neural networks later will be negatively affected.

In [6]:
```
from cs145.gradient_check import eval_numerical_gradient

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

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name]
, verbose=False)
    print('{} max relative error: {}'.format(param_name, rel_error(par
am_grad_num, grads[param_name])))
```

```
W2 max relative error: 8.800911222099066e-11
b2 max relative error: 2.4554844805570154e-11
W1 max relative error: 1.7476665046687833e-09
b1 max relative error: 7.382451041178829e-10
```

## Training the network

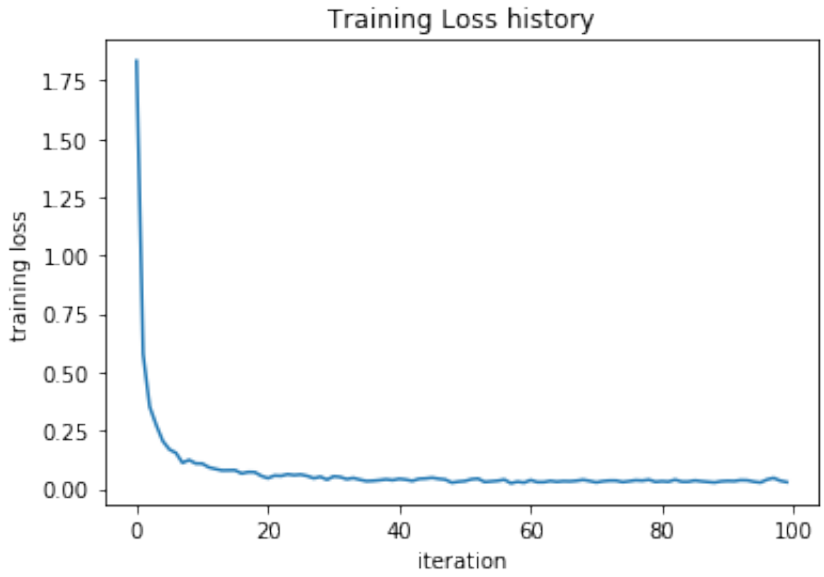
Implement neural_net.train() to train the network via stochastic gradient descent, much like the linear regression.

In [7]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.02950555626206818
```




# Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [8]:
```
from cs145.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 p
repare
    it for the two-layer neural net classifier. These are the same ste
ps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = './cs145/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,)
```

## Running SGD

If your implementation is correct, you should see a validation accuracy of around 15-18%.

In [9]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-5, learning_rate_decay=0.95,
            reg=0.1, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

# Save this net as the variable subopt_net for later comparison.
subopt_net = net
```

```
iteration 0 / 1000: loss 0.5000623457905098
iteration 100 / 1000: loss 0.4998246529435278
iteration 200 / 1000: loss 0.4995946718475304
iteration 300 / 1000: loss 0.49933536166627984
iteration 400 / 1000: loss 0.4989962372581251
iteration 500 / 1000: loss 0.49847178744773624
iteration 600 / 1000: loss 0.49758927830530253
iteration 700 / 1000: loss 0.4966248113033766
iteration 800 / 1000: loss 0.4958001901438695
iteration 900 / 1000: loss 0.4939583435911163
Validation accuracy:  0.172
```

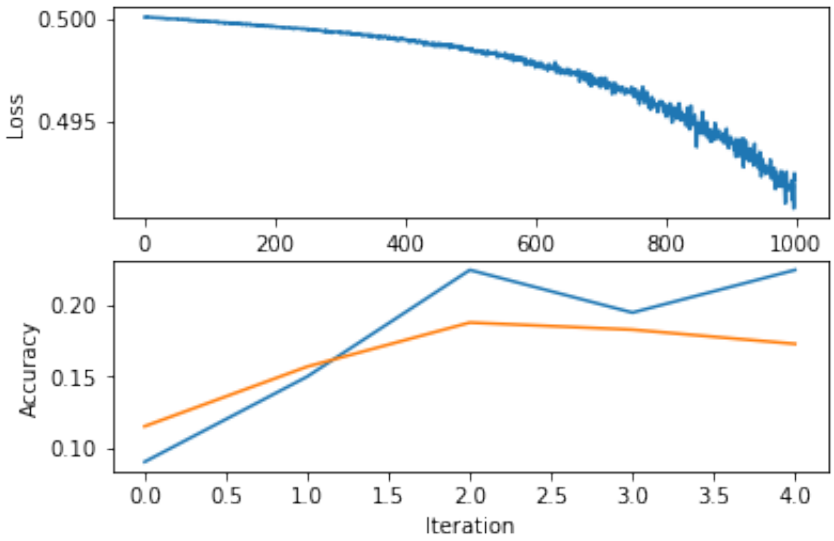
In [10]:
```
stats['train_acc_history']
```

Out[10]:
```
[0.09, 0.15, 0.225, 0.195, 0.225]
```

In [11]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')

plt.show()
```




## Questions:

The training accuracy isn't great. It seems even worse than simple KNN model, which is not as good as expected.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

### Answers:

(1)We need more iterations for the model to converge. learning rate is too small so that optimal value cannot be reached.

(2)Increase the number of iteration, increase learning rate or decrease learning rate decay.

---

## Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

**Important: Think about whether you should retrain a new model from scratch every time your try a new set of hyperparameters.**

In [12]:
```
best_net = None # store the best model into this

# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 45% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied
by:
#      min(floor((X - 23%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size
= 50)!
# ================================================================ #

# todo: optimal parameter search (you can use grid search by for-loops
)

input_size = 32 * 32 * 3 # do not change
hidden_size = 50 # do not change
num_classes = 10 # do not change
best_valacc = 0 # do not change

# Train the network and find best parameter:
best_net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = best_net.train(X_train, y_train, X_val, y_val, learning_rate=1
e-3, learning_rate_decay=0.8,
                       reg=1e-5,num_iters=5000, batch_size=100,verbose
=True)
best_valacc = (best_net.predict(X_val) == y_val).mean()

# Output your results
print("== Best parameter settings ==")
# print your best parameter setting here!
print("Best accuracy on validation set: {}".format(best_valacc))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
iteration 0 / 5000: loss 0.49995748789812644
iteration 100 / 5000: loss 0.4061266428812209
iteration 200 / 5000: loss 0.3871345353596667
iteration 300 / 5000: loss 0.3908208391880463
iteration 400 / 5000: loss 0.37495489249306907
iteration 500 / 5000: loss 0.367042968101788
iteration 600 / 5000: loss 0.3854009804130353
iteration 700 / 5000: loss 0.36647784580402754
```

```
iteration 800 / 5000: loss 0.3471370473421407
iteration 900 / 5000: loss 0.38577368911413273
iteration 1000 / 5000: loss 0.37122132395037416
iteration 1100 / 5000: loss 0.3621094525698837
iteration 1200 / 5000: loss 0.34759906894201514
iteration 1300 / 5000: loss 0.37807319093053415
iteration 1400 / 5000: loss 0.3441585350364839
iteration 1500 / 5000: loss 0.35923354511093114
iteration 1600 / 5000: loss 0.35709756150546484
iteration 1700 / 5000: loss 0.34831741538681504
iteration 1800 / 5000: loss 0.34490055544641646
iteration 1900 / 5000: loss 0.3733812631621005
iteration 2000 / 5000: loss 0.32676260899123283
iteration 2100 / 5000: loss 0.35625701293803286
iteration 2200 / 5000: loss 0.35167167931257987
iteration 2300 / 5000: loss 0.3373126854686802
iteration 2400 / 5000: loss 0.3465034445353386
iteration 2500 / 5000: loss 0.33349838633187834
iteration 2600 / 5000: loss 0.34331159844091197
iteration 2700 / 5000: loss 0.3484679533293963
iteration 2800 / 5000: loss 0.3341069565824827
iteration 2900 / 5000: loss 0.34697403261271154
iteration 3000 / 5000: loss 0.3473109304425691
iteration 3100 / 5000: loss 0.362061603580868
iteration 3200 / 5000: loss 0.3343963921392651
iteration 3300 / 5000: loss 0.33088946709123684
iteration 3400 / 5000: loss 0.34322050961980366
iteration 3500 / 5000: loss 0.36160225982846644
iteration 3600 / 5000: loss 0.31966185225136695
iteration 3700 / 5000: loss 0.3190673153144391
iteration 3800 / 5000: loss 0.3462496870664184
iteration 3900 / 5000: loss 0.3386779303021736
iteration 4000 / 5000: loss 0.34064990676350726
iteration 4100 / 5000: loss 0.3631226227422284
iteration 4200 / 5000: loss 0.3203329009250138
iteration 4300 / 5000: loss 0.3628094150760723
iteration 4400 / 5000: loss 0.3424642875801414
iteration 4500 / 5000: loss 0.32245760787991584
iteration 4600 / 5000: loss 0.3531293330725337
iteration 4700 / 5000: loss 0.3246658859709863
iteration 4800 / 5000: loss 0.3323532314924037
iteration 4900 / 5000: loss 0.33061480014240496
== Best parameter settings ==
Best accuracy on validation set: 0.498
```

### Quesions

(1) What is your best parameter settings? (Output from the previous cell)

(2) What parameters did you tune? How are they changing the performance of nerural network? You can discuss any observations from the optimization.

### Answers

(1)The following is my best parameter setting: learning_rate=1e-3, learning_rate_decay=0.8, reg=1e-5,num_iters=5000, batch_size=100

(2)More iteration makes the accuracy better, but it stops changing after 5000 iterations. Learning rate and learning decay is tricky because when they become too large or too small, opitmal solution may be skipped.

---

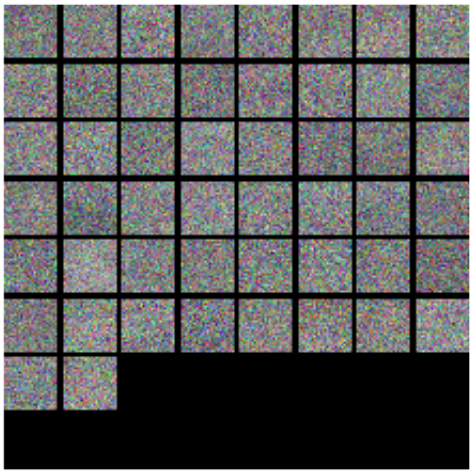
## Visualize the weights of your neural networks

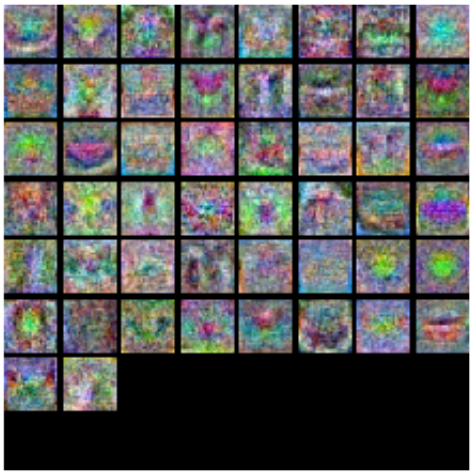
In [13]:
```
from cs145.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```





## Questions:

What differences do you see in the weights between the suboptimal net and the best net you arrived at? What do the weights in neural networks probably learn after training?

## Answer:

The suboptimal net does not seem to have meaningful result. However, the best net I have seem to give some meaning weights that can do classification.

# Evaluate on test set

In [14]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
#test_acc = (subopt_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.481
```

### Questions:

(1) What is your test accuracy by using the best NN you have got? How much does the performance increase compared with kNN? Why can neural networks perform better than kNN?

(2) Do you have any other ideas or suggestions to further improve the performance of neural networks other than the parameters you have tried in the homework?

### Answers:

(1) The Best accuracy is 0.481. It increased from 0.284 to 0.481. For this task, NN learns different high level features, so the result is more accurate. kNN finds neighbours for each pixel, which is not that accurate.

(2) Maybe use NN with more layers so we can detect more high level features

---

# Bonus Question: Change MSE Loss to Cross Entropy Loss

This is a bonus question. If you finish this (cross entropy loss) correctly, you will get **up to 20 points** (add up to your HW3 score).

Note: From grading policy of this course, your maximum points from homework are still 25 out of 100, but you can use the bonus question to make up other deduction of other assignments.

Pass output scores in networks from forward pass into softmax function. The softmax function is defined as,

$$p_j = \sigma(z_j) = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}}$$

After softmax, the scores can be considered as probability of $j$-th class.

The cross entropy loss is defined as,

$$L = L_{\text{CE}} + L_{reg} = \frac{1}{N} \sum_{i=1}^{N} \log\left(p_{i,j}\right) + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

To take derivative of this loss, you will get the gradient as,

$$\frac{\partial L_{\text{CE}}}{\partial o_i} = p_i - y_i$$

More details about multi-class cross entropy loss, please check http://cs231n.github.io/linear-classify/ (http://cs231n.github.io/linear-classify/) and more explanation (https://deepnotes.io/softmax-crossentropy) about the derivative of cross entropy.

Change the loss from MSE to cross entropy, you only need to change you `MSE_loss(x,y)` in `TwoLayerNet.loss()` function to `softmax_loss(x,y)`.

**Now you are free to use any code to show your results of the two-layer networks with newly-implemented cross entropy loss. You can use code from previous cells.**

In [15]:
```
# Start training your networks and show your results
```

In [18]:
```
input_size = 32 * 32 * 3 # do not change
hidden_size = 50 # do not change
num_classes = 10 # do not change
best_valacc = 0 # do not change

# Train the network and find best parameter:
best_net = TwoLayerNet(input_size, hidden_size, num_classes)
stats = best_net.train(X_train, y_train, X_val, y_val, learning_rate=1
e-3, learning_rate_decay=0.8,
                       reg=1e-5,num_iters=5000, batch_size=100,verbose
=True)
best_valacc = (best_net.predict(X_val) == y_val).mean()

# Output your results
print("== Best parameter settings ==")
# print your best parameter setting here!
print("Best accuracy on validation set: {}".format(best_valacc))
```

```
iteration 0 / 5000: loss 2.302572759808921
iteration 100 / 5000: loss 1.9472466586686816
iteration 200 / 5000: loss 1.7751336802291386
iteration 300 / 5000: loss 1.5449784128999153
iteration 400 / 5000: loss 1.6990033979119132
iteration 500 / 5000: loss 1.6212106333315355
iteration 600 / 5000: loss 1.6607164056072077
iteration 700 / 5000: loss 1.5301432090736347
iteration 800 / 5000: loss 1.4974181996474765
iteration 900 / 5000: loss 1.610152308853683
iteration 1000 / 5000: loss 1.370530957510399
iteration 1100 / 5000: loss 1.2572291355791498
iteration 1200 / 5000: loss 1.5929031620819636
iteration 1300 / 5000: loss 1.5125181153423608
iteration 1400 / 5000: loss 1.2877504222170575
iteration 1500 / 5000: loss 1.281561106767019
iteration 1600 / 5000: loss 1.2493551876458118
iteration 1700 / 5000: loss 1.5549912949226896
iteration 1800 / 5000: loss 1.6312724230929339
iteration 1900 / 5000: loss 1.4435793282134357
iteration 2000 / 5000: loss 1.5636971668784485
iteration 2100 / 5000: loss 1.2984110869364827
iteration 2200 / 5000: loss 1.3272347306578627
iteration 2300 / 5000: loss 1.3786203117949307
iteration 2400 / 5000: loss 1.4106690649146727
iteration 2500 / 5000: loss 1.3333974389627907
iteration 2600 / 5000: loss 1.3481386273731488
iteration 2700 / 5000: loss 1.3470513957516836
iteration 2800 / 5000: loss 1.444920709151591
iteration 2900 / 5000: loss 1.3179286318708554
iteration 3000 / 5000: loss 1.213482690356895
```

```
iteration 3100 / 5000: loss 1.4141133201728173
iteration 3200 / 5000: loss 1.3286419485440695
iteration 3300 / 5000: loss 1.2721482498976653
iteration 3400 / 5000: loss 1.325414010485897
iteration 3500 / 5000: loss 1.2838583718414094
iteration 3600 / 5000: loss 1.265853255782576
iteration 3700 / 5000: loss 1.4418870348667314
iteration 3800 / 5000: loss 1.3518955727326976
iteration 3900 / 5000: loss 1.5332612117350644
iteration 4000 / 5000: loss 1.2261314463724613
iteration 4100 / 5000: loss 1.216612771337736
iteration 4200 / 5000: loss 1.3918386253585662
iteration 4300 / 5000: loss 1.305196581994819
iteration 4400 / 5000: loss 1.311795326817284
iteration 4500 / 5000: loss 1.4768513841637025
iteration 4600 / 5000: loss 1.1702742267968358
iteration 4700 / 5000: loss 1.1456903008087709
iteration 4800 / 5000: loss 1.110879216949882
iteration 4900 / 5000: loss 1.2774143329119594
== Best parameter settings ==
Best accuracy on validation set: 0.507
```