This is the svm workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a linear support vector machine.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and includes code to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training an SVM classifier via gradient descent.

Importing libraries and data setup

In [1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-1
import pdb

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py fi
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [4]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500
# pdb.set_trace()

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]
```

```
# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[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]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

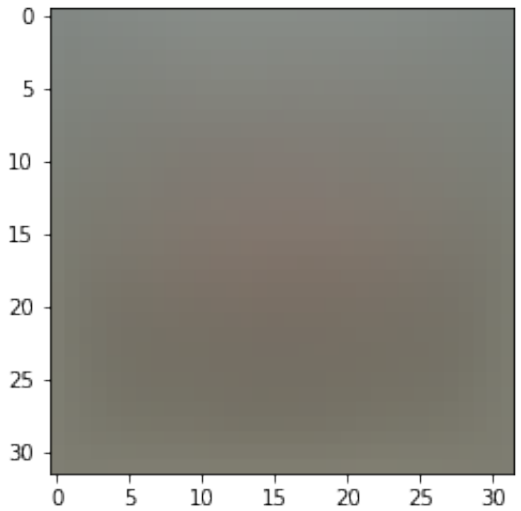
In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

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

In [6]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
# print(mean_image.shape) = (3072, )
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



In [7]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [8]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

Answer:

(1) Generally, we should normalize the data (or perform mean-subtraction on the data) when the scale of a feature is irrelevant or misleading, and not normalize when the scale is meaningful. As for SVM, we perform mean-subtraction to center the data, however for KNN, we use Euclidean distance to measure how far the testing data is from the training data, which indicates the scale is meaningful.

Training an SVM

The following cells will take you through building an SVM. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

In [31]:
```
from nndl.svm import SVM
```

In [32]:
```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial solutions consistent, we are going to us

np.random.seed(1)

num_classes = len(np.unique(y_train)) # = 10
num_features = X_train.shape[1] # = 3072

svm = SVM(dims=[num_classes, num_features])
```

SVM loss

In [33]:
```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.los
# tic = time.time()
loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))
# toc = time.time()
# print ('Naive loss and gradient: computed in %fs' % (toc - tic))
# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.97791541023.
```

SVM gradient

In [34]:
```
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = svm.loss_and_grad(X_dev,y_dev)
# print(X_dev[0][0])
# print(loss)
# print(grad)
# print(grad.shape)
# print(np.linalg.norm(grad, 'fro'))
# print(grad[0][0])
# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less i
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -11.246262 analytic: -11.246262, relative error: 2.249944e-
09
numerical: 5.186229 analytic: 5.186229, relative error: 1.630023e-09
numerical: -2.490384 analytic: -2.490384, relative error: 1.205736e-09
numerical: 4.578491 analytic: 4.578491, relative error: 1.036558e-08
numerical: -8.306767 analytic: -8.306767, relative error: 3.932551e-09
numerical: 4.852505 analytic: 4.852505, relative error: 7.428111e-10
numerical: 5.507388 analytic: 5.507389, relative error: 1.428190e-08
numerical: -14.231407 analytic: -14.231407, relative error: 2.042413e-
09
numerical: -7.068918 analytic: -7.068918, relative error: 3.936076e-09
numerical: -13.755380 analytic: -13.755380, relative error: 7.735600e-
09
```

A vectorized version of SVM

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [35]:
```
import time
```

In [36]:
```
## Implement svm.fast_loss_and_grad which calculates the loss and gradien
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.loss_and_grad(X_dev, y_dev)
# print(X_dev[0][0])
# print(grad.shape)
# print(grad[0][0])
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.

tic = time.time()
loss_vectorized, grad_vectorized = svm.fast_loss_and_grad(X_dev, y_dev)
# print(grad_vectorized.shape)
# print(grad_vectorized[0][0])
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vecto

# The losses should match but your vectorized implementation should be mu
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized,

# You should notice a speedup with the same output, i.e., differences on
```

```
Normal loss / grad_norm: 13999.630452635061 / 2048.18940097428 compute
d in 0.05036282539367676s
Vectorized loss / grad: 13999.630452635061 / 2048.1894009742796 comput
ed in 0.004767179489135742s
difference in loss / grad: 0.0 / 2.5154507727668504e-12
```

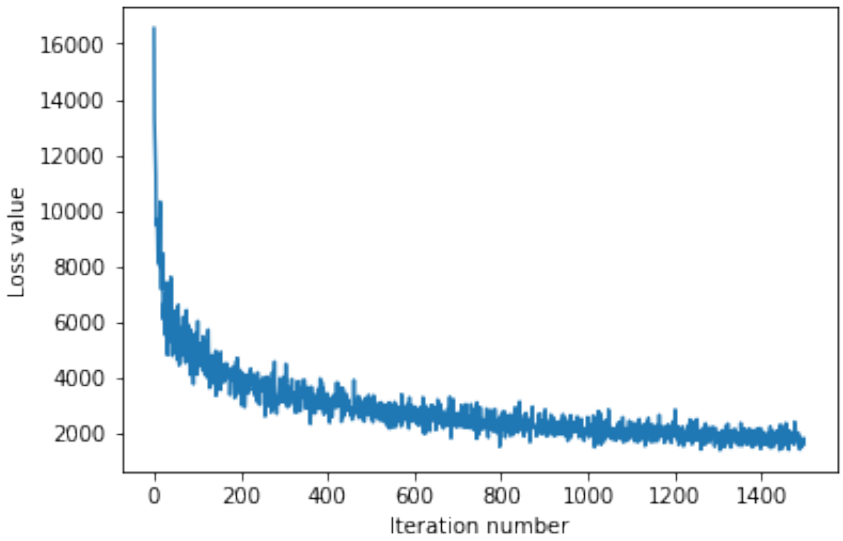
Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [37]:
```
# Implement svm.train() by filling in the code to extract a batch of data
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))
# print(len(loss_hist))
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 16557.380001909158
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942788
iteration 300 / 1500: loss 3681.9226471953625
iteration 400 / 1500: loss 2732.616437398899
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.0357842782664
iteration 700 / 1500: loss 2206.2348687399326
iteration 800 / 1500: loss 2269.03882411698
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.6921357268266
iteration 1100 / 1500: loss 2182.0689059051633
iteration 1200 / 1500: loss 1861.118224425044
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582114
That took 3.043776035308838s
```



Evaluate the performance of the trained SVM on the validation data.

In [38]:
```
## Implement svm.predict() and use it to compute the training and testing

y_train_pred = svm.predict(X_train)
# print(y_train_pred[:10], y_train[:10])
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pre
y_val_pred = svm.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred
```

```
training accuracy: 0.28530612244897957
validation accuracy: 0.3
```

Optimize the SVM

Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

In [40]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the SVM with different learning rates and evaluate on the
#     validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best VALIDATION accuracy corresponding to the best VALIDATION
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
#   Note: You do not need to modify SVM class for this section
# ================================================================ #
learning_rates = np.array([5e-6, 8e-6, 1e-5, 5e-5, 8e-5, 1e-4, 5e-4, 8e-4
# learning_rates = np.array([7e-7])
# print(lr[0])
best_svm = None
best_val_acc = -1
best_lr = 0

for lr in learning_rates:
    svm = SVM(dims=[num_classes, num_features])
    losses = svm.train(X_train, y_train, learning_rate = lr,
                       num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train
    y_val_pred = svm.predict(X_val)
    val_acc = np.mean(np.equal(y_val, y_val_pred))
```

```
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_

    if val_acc > best_val_acc:
        best_svm = svm
        best_val_acc = val_acc
        best_lr = lr
print('The best learning rate is {}'.format(lr, ))
print('The best validation accuracy is {} and its corresponding validatio

y_test_pred = best_svm.predict(X_test)
test_acc = np.mean(np.equal(y_test, y_test_pred))
print('testing accuracy: {}'.format(test_acc, ))



# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
iteration 0 / 1500: loss 15658.054145417169
iteration 100 / 1500: loss 14242.049657996204
iteration 200 / 1500: loss 12999.715353665571
iteration 300 / 1500: loss 10538.180112933615
iteration 400 / 1500: loss 10286.567991053184
iteration 500 / 1500: loss 11864.392203922114
iteration 600 / 1500: loss 8748.991554380966
iteration 700 / 1500: loss 10365.671096386988
iteration 800 / 1500: loss 8974.786133569609
iteration 900 / 1500: loss 8564.14627988812
iteration 1000 / 1500: loss 7841.7792116224955
iteration 1100 / 1500: loss 9025.290033030535
iteration 1200 / 1500: loss 9334.831641553408
iteration 1300 / 1500: loss 8184.371643932415
iteration 1400 / 1500: loss 7184.292310306516
training accuracy: 0.18755102040816327
validation accuracy: 0.206
iteration 0 / 1500: loss 16409.02482607994
iteration 100 / 1500: loss 17161.9866271389
iteration 200 / 1500: loss 11600.392953096214
iteration 300 / 1500: loss 12994.295693589236
iteration 400 / 1500: loss 9685.6131095873
iteration 500 / 1500: loss 9801.677952888918
iteration 600 / 1500: loss 9938.963082617583
iteration 700 / 1500: loss 8585.069789627989
iteration 800 / 1500: loss 8026.595887017292
iteration 900 / 1500: loss 8785.667490177657
iteration 1000 / 1500: loss 8520.481294442534
iteration 1100 / 1500: loss 9886.230559998778
iteration 1200 / 1500: loss 7176.0734298244815
```

```
iteration 1300 / 1500: loss 7643.173475522494
iteration 1400 / 1500: loss 8082.74433363298
training accuracy: 0.19208163265306122
validation accuracy: 0.165
iteration 0 / 1500: loss 16965.92017072196
iteration 100 / 1500: loss 13241.267181836445
iteration 200 / 1500: loss 9782.150495897133
iteration 300 / 1500: loss 9436.614217241558
iteration 400 / 1500: loss 9560.581023682797
iteration 500 / 1500: loss 9421.168462190435
iteration 600 / 1500: loss 9128.983403691296
iteration 700 / 1500: loss 9079.691635210442
iteration 800 / 1500: loss 6727.872424648836
iteration 900 / 1500: loss 7878.181874800411
iteration 1000 / 1500: loss 8319.263663070924
iteration 1100 / 1500: loss 7294.666145203879
iteration 1200 / 1500: loss 7025.540397134533
iteration 1300 / 1500: loss 6072.6697761872165
iteration 1400 / 1500: loss 5943.778634834087
training accuracy: 0.1990204081632653
validation accuracy: 0.219
iteration 0 / 1500: loss 17131.943492505638
iteration 100 / 1500: loss 9409.765300524099
iteration 200 / 1500: loss 7203.865319404643
iteration 300 / 1500: loss 6034.943598654187
iteration 400 / 1500: loss 6745.82004418117
iteration 500 / 1500: loss 5670.455158034281
iteration 600 / 1500: loss 5896.141137483634
iteration 700 / 1500: loss 5138.626257543199
iteration 800 / 1500: loss 4687.057624831159
iteration 900 / 1500: loss 5753.858369064131
iteration 1000 / 1500: loss 4893.345794452343
iteration 1100 / 1500: loss 4559.692621283092
iteration 1200 / 1500: loss 5127.9571601601
iteration 1300 / 1500: loss 4620.760032907103
iteration 1400 / 1500: loss 4538.825331754093
training accuracy: 0.24283673469387754
validation accuracy: 0.239
iteration 0 / 1500: loss 15053.531771301045
iteration 100 / 1500: loss 8548.27538426734
iteration 200 / 1500: loss 6430.751154991212
iteration 300 / 1500: loss 6266.1164642258245
iteration 400 / 1500: loss 4599.613637503997
iteration 500 / 1500: loss 5974.490158454784
iteration 600 / 1500: loss 4660.5231430064605
iteration 700 / 1500: loss 4985.933388183541
iteration 800 / 1500: loss 3825.8806909438044
iteration 900 / 1500: loss 4419.375276787923
iteration 1000 / 1500: loss 3778.805647728728
iteration 1100 / 1500: loss 4314.161554437147
```

```
iteration 1200 / 1500: loss 4178.345836798539
iteration 1300 / 1500: loss 3883.6383150637657
iteration 1400 / 1500: loss 3723.6860896329654
training accuracy: 0.2616734693877551
validation accuracy: 0.289
iteration 0 / 1500: loss 15486.370619706
iteration 100 / 1500: loss 7598.664926819008
iteration 200 / 1500: loss 6464.694178830311
iteration 300 / 1500: loss 6573.470184852069
iteration 400 / 1500: loss 4488.957222967042
iteration 500 / 1500: loss 5312.264469758165
iteration 600 / 1500: loss 5008.3494014087855
iteration 700 / 1500: loss 4226.553441522881
iteration 800 / 1500: loss 4008.37078310916
iteration 900 / 1500: loss 3920.2236802780444
iteration 1000 / 1500: loss 3459.0430244524714
iteration 1100 / 1500: loss 4159.028161737634
iteration 1200 / 1500: loss 3717.538592912142
iteration 1300 / 1500: loss 3271.164998538439
iteration 1400 / 1500: loss 3855.4419267411267
training accuracy: 0.26193877551020406
validation accuracy: 0.262
iteration 0 / 1500: loss 14115.646035437041
iteration 100 / 1500: loss 5016.086890251574
iteration 200 / 1500: loss 3916.7498368166257
iteration 300 / 1500: loss 3360.551087726868
iteration 400 / 1500: loss 3314.097752815921
iteration 500 / 1500: loss 2921.0202953081916
iteration 600 / 1500: loss 2779.798949464384
iteration 700 / 1500: loss 2831.2160545899287
iteration 800 / 1500: loss 2068.6456613162136
iteration 900 / 1500: loss 2339.112428000442
iteration 1000 / 1500: loss 1779.0262248358026
iteration 1100 / 1500: loss 1765.5903230297836
iteration 1200 / 1500: loss 1971.8541271736024
iteration 1300 / 1500: loss 1830.2517370712987
iteration 1400 / 1500: loss 1618.720837519034
training accuracy: 0.28681632653061223
validation accuracy: 0.283
iteration 0 / 1500: loss 15500.773674459997
iteration 100 / 1500: loss 3624.5922360114073
iteration 200 / 1500: loss 4343.961495827747
iteration 300 / 1500: loss 3560.7895022717144
iteration 400 / 1500: loss 2805.5071394041374
iteration 500 / 1500: loss 2306.8103829786837
iteration 600 / 1500: loss 2224.9232990383807
iteration 700 / 1500: loss 2188.916391473069
iteration 800 / 1500: loss 2266.7133723585125
iteration 900 / 1500: loss 2332.4208940134804
iteration 1000 / 1500: loss 2162.282582603412
```

```
iteration 1100 / 1500: loss 1771.2063745628884
iteration 1200 / 1500: loss 1981.022152186711
iteration 1300 / 1500: loss 1644.920880524143
iteration 1400 / 1500: loss 1817.3431195249425
training accuracy: 0.28753061224489795
validation accuracy: 0.291
iteration 0 / 1500: loss 14366.939852858146
iteration 100 / 1500: loss 4097.419944971674
iteration 200 / 1500: loss 4169.394931893793
iteration 300 / 1500: loss 2959.2867961443844
iteration 400 / 1500: loss 2812.2811461175006
iteration 500 / 1500: loss 2057.166534030488
iteration 600 / 1500: loss 2302.0978656105453
iteration 700 / 1500: loss 2208.333129706169
iteration 800 / 1500: loss 2577.666764598992
iteration 900 / 1500: loss 2118.8357795914653
iteration 1000 / 1500: loss 2188.4634790977493
iteration 1100 / 1500: loss 2157.654618869853
iteration 1200 / 1500: loss 1957.6248272100063
iteration 1300 / 1500: loss 1941.4776528681127
iteration 1400 / 1500: loss 1851.4610202592694
training accuracy: 0.29475510204081634
validation accuracy: 0.286
iteration 0 / 1500: loss 17376.25774264116
iteration 100 / 1500: loss 8478.345173905172
iteration 200 / 1500: loss 9984.016614609951
iteration 300 / 1500: loss 10648.755516124982
iteration 400 / 1500: loss 11807.343750031527
iteration 500 / 1500: loss 7923.3802144688625
iteration 600 / 1500: loss 7964.524670984784
iteration 700 / 1500: loss 8457.783570224037
iteration 800 / 1500: loss 11117.136942419613
iteration 900 / 1500: loss 7942.514939660153
iteration 1000 / 1500: loss 9728.265369219092
iteration 1100 / 1500: loss 7475.149334345299
iteration 1200 / 1500: loss 8774.045743536513
iteration 1300 / 1500: loss 10006.357867329512
iteration 1400 / 1500: loss 10995.244866043186
training accuracy: 0.3222857142857143
validation accuracy: 0.31
iteration 0 / 1500: loss 13671.730311763002
iteration 100 / 1500: loss 13568.61472717571
iteration 200 / 1500: loss 18858.707093058416
iteration 300 / 1500: loss 15017.082958133842
iteration 400 / 1500: loss 9663.565128809954
iteration 500 / 1500: loss 20711.68307613329
iteration 600 / 1500: loss 21258.101155363034
iteration 700 / 1500: loss 10449.750647941044
iteration 800 / 1500: loss 14288.761509318589
iteration 900 / 1500: loss 16999.314728781927
```

```
iteration 1000 / 1500: loss 9742.530917681514
iteration 1100 / 1500: loss 15853.653217545927
iteration 1200 / 1500: loss 17213.626105314437
iteration 1300 / 1500: loss 16620.277121523613
iteration 1400 / 1500: loss 10166.822256093983
training accuracy: 0.2734489795918367
validation accuracy: 0.27
The best learning rate is 0.008
The best validation accuracy is 0.31 and its corresponding validation
error is 0.69
testing accuracy: 0.312
```

In [ ]: