Assignment 1 Report

심명진

Q1: k-Nearest Neighbor classifier

Q2: Training a Support Vector Machine

Q3: Implement a Softmax classifier

Q4: Two-Layer Neural Network

Q5: Higher Level Representations: Image Features

[FULL CODE] assignment 1

https://github.com/myeongmy/deepLearning/tree/master/assignment1

Q1 : k-Nearest Neighbor classifier

Knn 분류기는 다른 선형 분류기나 신경망 네트워크, CNN과는 다르게 데이터 중심 접근방법을 이용한다. 각 카테고리 별로 이미지들을 수집하여 이미지와 라벨 자체를 모델에 훈련시킨다. 훈련 시간은 적게 드는 반면, 새로운 이미지에 대한 prediction이 오래 걸린다는 단점이 있다.

➔ 이 과제에서 구현해야 할 것

1. Image classification 과정 이해
2. 하이퍼파라미터인 k를 구하기 위한 cross-validation 시행

<knn.ipynb>

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/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,)
```

먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에 대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.

```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]
```

```
# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

이 예제에서는 training data로 5000개, testing data로 500개만 이용하기로 한다. 그리고 각 이미지 데이터를 32*32*3 열로 길게 펼친다.

<k_nearest_neighbor.py>

Training data를 k_nearest_neighbor.py의 train 함수로 훈련시킨 뒤, 500개의 testing data와 5000개의 training data 간의 거리 비교를 하려고 한다.

먼저 두 개의 반복 loop를 이용한 방법이다.

```
def compute_distances_two_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using a nested loop over both the training data and the
  test data.

  Inputs:
  - X: A numpy array of shape (num_test, D) containing test data.

  Returns:
  - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    is the Euclidean distance between the ith test point and the jth training
    point.
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  for i in xrange(num_test):
    for j in xrange(num_train):
      dists[i,j] = np.sqrt(np.sum(np.square(self.X_train[j] - X[i,:])))

      #########################################################################
      # TODO:                                                                 #
      # Compute the l2 distance between the ith test point and the jth        #
      # training point, and store the result in dists[i, j]. You should       #
      # not use a loop over dimension.                                        #
      #########################################################################

      #########################################################################
      #                       END OF YOUR CODE                                #
      #########################################################################
  return dists
```

input으로 testing data X를 받아 testing data의 개수를 num_test 변수에 저장하고 인스턴스 변수인 X_train을 통해 training data의 개수를 num_train에 저장한다. Dists 배열을 (500, 5000) shape의 모든 원소가 0인 배열로 초기화를 한 후, 각 for문을 돌면서 인덱스 하나하나씩 채워간다. 우리는 L2 distance를 구해야 하므로 500개의 testing data를 모든 training data와 한 번씩 비교를 해나가는데 각 셀 값의 차를 제곱하고 그 값들을 전부 더하여 루트 연산을 적용하면 해당 training data와 해당 testing data의 거리 차이가 계산된다. 그리고 그 dists 배열을 리턴한다.

```
def predict_labels(self, dists, k=1):
  """
  Given a matrix of distances between test points and training points,
  predict a label for each test point.

  Inputs:
  - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    gives the distance betwen the ith test point and the jth training point.

  Returns:
  - y: A numpy array of shape (num_test,) containing predicted labels for the
    test data, where y[i] is the predicted label for the test point X[i].
  """
  num_test = dists.shape[0]
  y_pred = np.zeros(num_test)
  for i in xrange(num_test):
    # A list of length k storing the labels of the k nearest neighbors to
    # the ith test point.
    closest_y = []
    #########################################################################
    # TODO:                                                                 #
    # Use the distance matrix to find the k nearest neighbors of the ith    #
    # testing point, and use self.y_train to find the labels of these       #
    # neighbors. Store these labels in closest_y.                           #
    # Hint: Look up the function numpy.argsort.                             #
    #########################################################################
    dists_sort = np.argsort(dists[i,:])
    closest_y = self.y_train(dists_sort[0:k])

    #########################################################################
    # TODO:                                                                 #
    # Now that you have found the labels of the k nearest neighbors, you    #
    # need to find the most common label in the list closest_y of labels.   #
    # Store this label in y_pred[i]. Break ties by choosing the smaller     #
    # label.                                                                #
    #########################################################################
    classes, countNum = np.unique(closest_y, return_counts=True)
    y_pred[i]=classes(np.argmax(countNum))
    #########################################################################
    #                           END OF YOUR CODE                            #
    #########################################################################
```

위에서 구한 dists 배열을 가지고 predict_labels 함수를 이용해 testing data에 대한 라벨 값을 예측한다. Testing data 500개 하나하나 for문을 돌면서 예측되는 라벨을 구해 y_pred 배열을 완성한다. Closest_y에는 5000개의 training data와의 거리 데이터 중 가장 작은 거리에 해당하는 k개를 골라 그에 해당하는 라벨들을 저장한다. 그리고 그 closest_y에 unique 연산을 실시해 각 클래스(라벨)별로 개수를 구한다. 그래서 개수가 가장 큰 클래스(라벨)을 해당 testing data의 라벨 값으로 최종 결정한다.

이제는 거리 측정 연산의 속도를 높이기 위해 loop를 한 번만 사용하기로 한다.

```
def compute_distances_one_loop(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using a single loop over the test data.

  Input / Output: Same as compute_distances_two_loops
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  for i in xrange(num_test):
      dists[i, :] = np.sqrt(np.sum(np.square(self.X_train - X[i, :]), axis = 1))
    #######################################################################
    # TODO:                                                               #
    # Compute the l2 distance between the ith test point and all training #
    # points, and store the result in dists[i, :].                        #
    #######################################################################

    #######################################################################
    #                         END OF YOUR CODE                            #
    #######################################################################
  return dists
```

Loop는 testing data 500개에 대해서만 하나 돌며 dists[i]를 각각 구한다. Numpy는 vectorized 연산을 제공하므로 self.X_train – X[I]를 하면 각 행별로 뺄셈을 한 결과가 나오게 되고 그것을 제곱하여 axis = 1로 덧셈을 하면 각 행별로 덧셈이 이루어진다. 그래서 그것에 루트 연산을 해주게 되면 하나의 testing data에 대한 거리 측정값을 구하게 된다.

➔ 결과 : loop를 두 개를 이용하나 하나를 이용하나 거리 측정 값에는 변화가 없음을 알 수 있다.

```
]: # Now lets speed up distance matrix computation by using partial vectorization
   # with one loop. Implement the function compute_distances_one_loop and run the
   # code below:
   dists_one = classifier.compute_distances_one_loop(X_test)

   # To ensure that our vectorized implementation is correct, we make sure that it
   # agrees with the naive implementation. There are many ways to decide whether
   # two matrices are similar; one of the simplest is the Frobenius norm. In case
   # you haven't seen it before, the Frobenius norm of two matrices is the square
   # root of the squared sum of differences of all elements; in other words, reshape
   # the matrices into vectors and compute the Euclidean distance between them.
   difference = np.linalg.norm(dists - dists_one, ord='fro')
   print('Difference was: %f' % (difference, ))
   if difference < 0.001:
       print('Good! The distance matrices are the same')
   else:
       print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

마지막으로 loop를 아예 사용하지 않고 fully vectorized하게 거리 측정 함수를 구현해 보면

```
def compute_distances_no_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using no explicit loops.

  Input / Output: Same as compute_distances_two_loops
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  #########################################################################
  # TODO:                                                                 #
  # Compute the l2 distance between all test points and all training      #
  # points without using any explicit loops, and store the result in      #
  # dists.                                                                #
  #                                                                       #
  # You should implement this function using only basic array operations; #
  # in particular you should not use functions from scipy.                #
  #                                                                       #
  # HINT: Try to formulate the l2 distance using matrix multiplication    #
  #       and two broadcast sums.                                         #
  #########################################################################
  X_2 = np.sum(X**2, axis=1)     # row
  X_train_2 = np.sum(self.X_train**2, axis = 1)
  XX_train = np.dot(X,self.X_train.T)

  dists = np.sqrt(X_2[:,np.newaxis] -2*XX_train + X_train_2)
  #########################################################################
  #                         END OF YOUR CODE                              #
  #########################################################################
  return dists
```

Loop를 쓰지 않고 구현하기 위해 l2 distance를 구하는 공식, 즉 완전제곱식의 형태를 풀어서 계산해보았다. 따라서, testing data의 제곱 값을 따로 구하고, training data의 제곱 값을 따로 구하고, 그 둘의 내적 곱을 따로 구해 마지막에 덧/뺄셈을 통해 dists를 구하였다.

➔ 결과 : 이것도 마찬가지로 앞에서 구한 two loop를 이용해서 구하는 방식과 결과 값은 다르지 않았다. 하지만 실행 시간 면에서는 훨씬 더 빠른 속도를 보였다. (왜 one loop가 더 오래 걸리는지 모르겠습니다..)

```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# you should see significantly faster performance with the fully vectorized implementation
```

```
Two loop version took 29.739395 seconds
One loop version took 45.768689 seconds
No loop version took 0.544868 seconds
```

Cross-validation (좋은 k를 선정하기 위함)

<knn.ipynb>

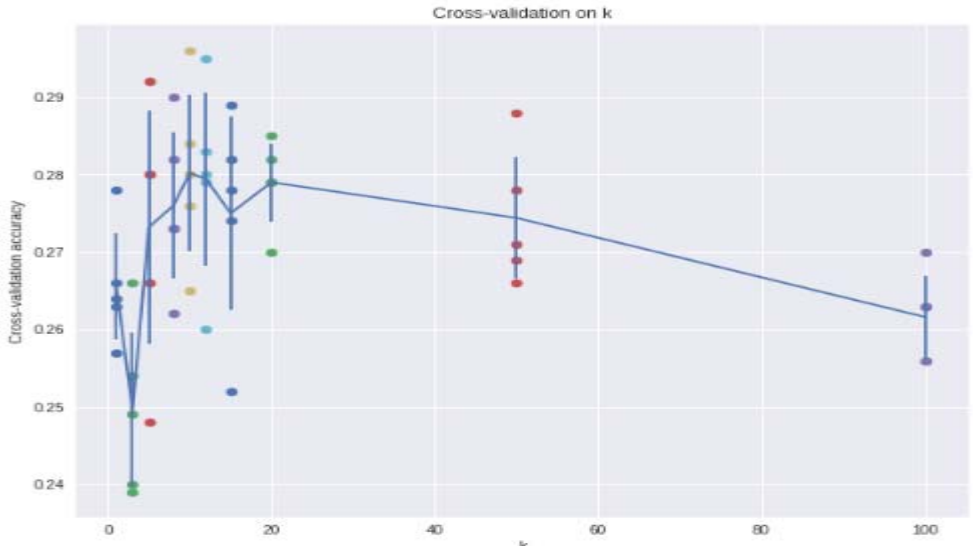
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
X_train_folds = np.array_split(X_train, 5)
y_train_folds = np.array_split(y_train, 5)
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

Fold의 개수는 5개라고 하고 training data를 다섯 등분해준다. Array_split 연산을 이용하여 나눠주고 각각 X_train_folds, y_train_folds 배열에 담는다.

```
k_to_accuracies = {}


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
for k in k_choices :
    for m in range(num_folds) :
        train_folds_X = np.concatenate(tuple([X_train_folds[i] for i in range(num_folds) if i != m]))
        train_folds_y = np.concatenate(tuple([y_train_folds[i] for i in range(num_folds) if i != m]))

        test_fold_X = X_train_folds[m]
        test_fold_y = y_train_folds[m]

        classifier.train(train_folds_X, train_folds_y)
        y_predict = classifier.predict(test_fold_X, k=k)

        val_correct = np.sum(y_predict == test_fold_y)

        if k not in k_to_accuracies :
            k_to_accuracies[k] = [val_correct / test_fold_X.shape[0]]
        else :
            k_to_accuracies[k].append(val_correct / test_fold_X.shape[0])
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

모든 k의 후보에 대해 knn 알고리즘을 적용하고 그 중 정확도가 제일 좋았던 k를 선택한다. 각 k마다 cross-validation 기법으로 훈련시키고 테스트 하기 때문에 하나의 k에 대하여 정확도 값은 5개가 나온다. (5개의 fold로 나누었으므로 각 fold가 한 번씩 testing data가 되는 것이다)

➔ 결과 : k=10일 때 정확도가 가장 좋아보인다.



따라서, k=10이라 하고 최종적으로 훈련시키고 정확도를 다시 계산하여보면 28.2%가 나온다.

```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

Q2: Training a Support Vector Machine

SVM classifier는 앞의 knn classifier와 다르게 데이터 중심의 접근 방법이 아닌 parametric 접근 방식이다. Training data를 이용하여 가중치 w를 최적화하여 이미지 분류를 가능하게 한다.

➔ 이 과제에서 구현해야 할 것

1. Loss function과 그에 대한 gradient 구하기
2. 최적의 learning rate와 regularization strength 찾기
3. SGD 방법을 이용하여 손실 함수 최적화

<svm.ipynb>

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/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,)
```

먼저 데이터셋(CIFAR-10)을 로드하여 training data와 그에 대한 라벨 데이터, testing data와 그에

대한 라벨 데이터를 마련한다. Training data는 50000개, testing data는 10000개에 해당한다.

```
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, 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,)
```

```
# 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)
```

데이터를 training, validation, testing set 세 가지로 나눈다. 그리고 training data 중 일부를 development set으로 가진다. 그리고 각 데이터셋들을 32*32*3 열로 길게 펼쳐준다.

```
# 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
```

```
# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# 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)
```

Training data를 모델에 훈련시키기 전에 데이터 전처리 과정이 필요하다. Training data와 testing data로부터 mean image를 추출하여 빼주고 bias 항을 추가해준다.

데이터 전처리가 완료되면 이제 svm loss function을 구현해야한다. 첫 번째는 loss function을 vectorized 연산을 이용하지 않고 naive하게 구현한다. 가중치 W는 작은 수로 초기화시켜두고 loss를 계산한다.

<linear_svm.py>

Def svm_loss_naive

```
"""
dW = np.zeros(W.shape) # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
num_class_greater_margin = 0
for i in xrange(num_train):
  scores = X[i].dot(W)
  correct_class_score = scores[y[i]]
  for j in xrange(num_classes):
    if j == y[i]:
      continue
    margin = scores[j] - correct_class_score + 1 # note delta = 1
    if margin > 0:
          num_class_greater_margin += 1
          loss += margin
          dW[:,j] += X[i,:]       # gradient for non correct class

  dW[:,y[i]] -= X[i,:]*num_class_greater_margin

# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)

dW = dW / num_train + 2*reg*W
```

함수의 입력으로 가중치 w와 데이터와 데이터 라벨 배열을 받으면 각 이미지 데이터 별로 데이터 X와 W의 내적 곱을 구하여 각 클래스에 대한 score를 얻어낸다. 그리고 해당 이미지의 정답 클래스에 해당하는 점수를 따로 저장해둔 뒤, 모든 다른 클래스와 해당 점수를 비교하여 마진 값을 구한다. 이 때, 모든 클래스를 for문을 돌면서 정답 클래스에 해당하는 경우는 따로 마진 값을 계산하지 않고 넘어가고 그 이외의 클래스에 대해서는 마진 값을 계산한 뒤 해당 마진 값이 0보다 큰 경우에만 loss에 그 값을 더해준다. 이것이 svm loss를 구하는 방식이다. 이렇게 총 loss를 구해준 뒤 training data의 개수로 나눠주어 평균 loss를 얻고 마지막에 l2 regularization을 한다는

가정하에 regularization 항을 추가해준다.

Gradient 같은 경우에는 우리가 가중치 W에 대한 gradient가 필요한 것이므로 loss 계산해주었던 수식을 토대로 차례차례 편미분을 통해 식을 얻어낼 수 있다. 앞서 loss를 구할 때 마진 값이 0보다 큰 경우에만 loss에 해당 마진 값을 더해주었으므로 dW에도 그 경우에만 X[i] 값을 더해주고 정답 클래스의 경우에는 마진 값이 0 이상이었던 클래스의 개수만큼 오히려 빼준다. 그리고 전체 loss를 training data의 개수로 나누어주었던 것처럼 dW도 똑같이 나누어주고, reg*np.sum(W*W)의 미분 값인 2*reg*W도 추가로 더해준다.

다음은 똑같은 svm loss를 vectorized하게 구현한다.

Def svm_loss_vectorized

```
    loss = 0.0
    dW = np.zeros(W.shape) # initialize the gradient as zero
    num_classes = W.shape[1]
    num_train = X.shape[0]
    #############################################################################
    # TODO:                                                                     #
    # Implement a vectorized version of the structured SVM loss, storing the    #
    # result in loss.                                                           #
    #############################################################################
    #Calculating the score
    scores = X.dot(W)
    Y = np.zeros(scores.shape)
    for i,row in enumerate(Y):
      row[y[i]] = 1

    Correct_class_scores = np.array( [ [ scores[i][y[i]] ]*num_classes for i in range(num_train) ] )
    Margin = scores - Correct_class_scores + ((scores - Correct_class_scores) != 0)
    X_with_margin_count = np.multiply(X.T , ( Margin > 0).sum(1) ).T

    loss += np.sum((Margin>0)*Margin)/num_train
    loss += 0.5 * reg * np.sum(W * W)
    dW += ( Margin > 0 ).T.dot(X).T/num_train
    dW -= (Margin == 0).T.dot(X_with_margin_count).T/num_train
    dW += reg*W
    #############################################################################
    #                             END OF YOUR CODE                              #
    #############################################################################
```

앞의 naive하게 구현한 것과 로직은 같다. 다만 앞에서는 for문을 돌며 각 testing image마다 각각 loss를 구하여 더해주었다면 이번에는 numpy에서 제공하는 vectorized 연산을 이용하여 한 번에 전체 testing images에 대한 margin 값을 구하고 loss를 구한다. 반복하지 않고 한 번에 loss를 계산하므로 이 방법이 실행 시간 면에서 더 빠르다.

```
[12]: # Next implement the function svm_loss_vectorized; for now only compute the loss;
      # we will implement the gradient in a moment.
      tic = time.time()
      loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
      toc = time.time()
      print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

      from cs231n.classifiers.linear_svm import svm_loss_vectorized
      tic = time.time()
      loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
      toc = time.time()
      print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

      # The losses should match but your vectorized implementation should be much faster.
      print('difference: %f' % (loss_naive - loss_vectorized))

Naive loss: 9.142981e+00 computed in 0.770137s
Vectorized loss: 9.142981e+00 computed in 0.503109s
difference: 0.000000
```

실제로 naive loss와 vectorized loss는 측정된 값은 같지만 계산하기 위해 걸린 시간은 다른 것을 알 수 있다.

SGD(Stochastic Gradient Descent) – 가중치 W를 최적화하여 loss값을 최소화하기 위한 알고리즘

<linear_classifier.py>

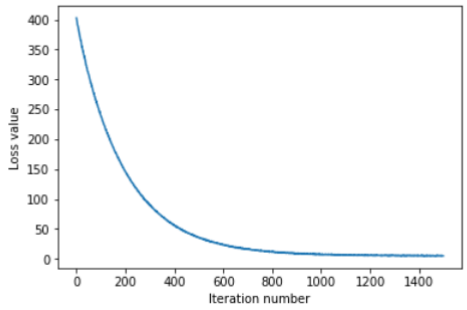
```
40      loss_history = []
41      for it in xrange(num_iters):
42        X_batch = None
43        y_batch = None
44
45        #########################################################################
46        # TODO:                                                                 #
47        # Sample batch_size elements from the training data and their           #
48        # corresponding labels to use in this round of gradient descent.        #
49        # Store the data in X_batch and their corresponding labels in           #
50        # y_batch; after sampling X_batch should have shape (dim, batch_size)   #
51        # and y_batch should have shape (batch_size,)                           #
52        #                                                                       #
53        # Hint: Use np.random.choice to generate indices. Sampling with         #
54        # replacement is faster than sampling without replacement.              #
55        #########################################################################
56        random_index = np.random.choice(list(range(num_train)), batch_size)
57        X_batch = X[random_index]
58        y_batch = y[random_index]
59        #########################################################################
60        #                       END OF YOUR CODE                                #
61        #########################################################################
62
63        # evaluate loss and gradient
64        loss, grad = self.loss(X_batch, y_batch, reg)
65        loss_history.append(loss)
66
67        # perform parameter update
68        #########################################################################
69        # TODO:                                                                 #
70        # Update the weights using the gradient and the learning rate.          #
71        #########################################################################
72        self.W += - learning_rate * grad
73        #########################################################################
74        #                       END OF YOUR CODE                                #
```

다음은 train() 함수의 일부를 캡쳐한 것이다. SGD 방법을 이용하여 훈련시키며 최적의 W를 찾아가는 과정인데 여기서는 mini-batch SGD를 이용하기로 한다. 이는 training data가 많으면 계속 반복하면서 계산을 하는데 시간이 너무 오래걸리므로 매 반복마다 랜덤으로 training data 중 일부를 뽑아서 그에 대한 loss를 계산하는 식으로 한다. 매 시행(반복)마다 랜덤하게 데이터를 추출하

므로 이론적으로는 mini-batch SGD를 수행한 결과와 SGD를 수행한 결과는 크게 다르지 않다. 함수의 입력으로 training data X, 그에 대한 라벨 y, 반복 횟수 등을 받아 실행을 하게 된다. Training data에서 batch_size만큼의 데이터를 추출하는 것은 np.random.choice() 연산을 이용한다. Training data의 인덱스 중 batch_size만큼 랜덤하게 추출한 후 해당 인덱스를 이용하여 X_batch, y_batch 배열을 만든다. 이제 이 데이터를 이용해 loss와 gradient를 계산한다. 계산된 loss는 loss가 줄어드는 과정을 저장하기 위해 loss_history 배열에 추가하고, 계산된 gradient를 이용해서는 가중치 W를 업데이트한다.

➔ 결과: 반복이 시행되면서 점점 loss가 줄어드는 것을 볼 수 있다.

```
In [18]: # A useful debugging strategy is to plot the loss as a function of
         # iteration number:
         plt.plot(loss_hist)
         plt.xlabel('Iteration number')
         plt.ylabel('Loss value')
         plt.show()
```



이제 훈련된 linear classifier를 가지고 새로운 testing data에 대해 라벨 예측을 시도한다.

```
def predict(self, X):
  """
  Use the trained weights of this linear classifier to predict labels for
  data points.

  Inputs:
  - X: A numpy array of shape (N, D) containing training data; there are N
    training samples each of dimension D.

  Returns:
  - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
  """
  y_pred = np.zeros(X.shape[0])
  ###########################################################################
  # TODO:                                                                   #
  # Implement this method. Store the predicted labels in y_pred.            #
  ###########################################################################
  scores_pred = X.dot(self.W)
  y_pred = np.argmax(scores_pred, axis = 1)
  ###########################################################################
  #                           END OF YOUR CODE                              #
  ###########################################################################
  return y_pred
```

Input data와 가중치 W를 내적 곱하여 각 클래스에 대한 score값인 scores_pred 배열을 얻는다. Scores_pred의 shpae는 (data 개수, 클래스 수)가 될 것이다. 각 data별로 가장 높은 점수를 얻은 클래스를 구해야 하므로 np.argmax() 연산을 이용한다. 그래서 최종적으로 y_pred(각 testing data

에 대한 라벨 예측값)을 리턴한다.

하이퍼파라미터로 가중치 W가 해당 모델의 성능을 좌지우지하는 가장 중요한 요소이기는 하지만, 또 다른 하이퍼파라미터인 regularization strength, learning rate 등도 잘 결정해야 한다. 실험을 해보기 위해 몇 가지 learning rate와 regularization_strength를 가지고 여러 번 훈련시켜보며 최적의 하이퍼파라미터를 찾아본다.

```
################################################################################
for i in range(2) :
    for j in range(2) :
        lc = LinearSVM()
        lc.train(X_train, y_train, learning_rate=learning_rates[i], reg=regularization_strengths[j],
                    num_iters=500, verbose=False)


        y_train_pred = lc.predict(X_train)



        y_val_pred = lc.predict(X_val)
        results[(learning_rates[i], regularization_strengths[j])] = (np.mean(y_train == y_train_pred), np.mean(y_va
l == y_val_pred))
        if np.mean(y_val == y_val_pred) > best_val :
              best_val = np.mean(y_val == y_val_pred)
              best_svm = lc


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

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.328694 val accuracy: 0.332000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.360388 val accuracy: 0.382000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.184571 val accuracy: 0.175000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.076939 val accuracy: 0.069000
best validation accuracy achieved during cross-validation: 0.382000
```

linearSVM을 하이퍼파라미터를 약간씩 조정하여 총 4번 훈련시킨다. 그리고 predict() 함수를 이용하여 각각 training data에 대한 예측 값과 validation data에 대한 예측 값을 구하여 분류기의 정확도를 구해본다. 가장 정확도가 높았던 모델과 정확도 값을 마지막에 출력하기 위하여 best_val과 best_svm에 저장한다.

In [32]:
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.356000
```

위에서 가장 성능이 좋았던 모델을 사용하여 최종적으로 testing set을 가지고 예측을 해본 결과 35.6%의 정확도를 보였다.

Q3: Implement a Softmax classifier

Softmax classifier는 앞에서 구현한 svm classifier와 같은 linear classifier이다. 다만 다른 점은 손실 함수를 구하는 방법이 다르다는 것이다. Svm classifier는 정답 클래스의 점수 값과 얼마나 큰 차이를 보이느냐에 초점을 둔 반면, softmax classifier는 전체 클래스에 대한 정답 클래스의 확률 값을 계산한다.

➔ 이 과제에서 구현해야 할 것

1. Softmax loss function을 구현하고 그에 대한 gradient 구하기
2. Validation set을 이용하여 hyperparameter 조정하기
3. SGD를 이용하여 loss function 최적화

(앞의 예제들과 dataset들의 shape와 값이 동일하므로 데이터 로드하는 부분은 생략한다.)

Softmax loss function을 두 가지 방법으로 구현한다. 첫 번째는 vectorized 연산을 사용하지 않고 naive하게 구현해본다.

<softmax.py>

Def softmax_loss_naive

```
32    ##############################################################################
33
34    num_classes = W.shape[1]
35    num_train = X.shape[0]
36
37    scores = X.dot(W)
38
39
40    for ii in range(num_train):
41      current_scores = scores[ii, :]
42
43
44      corrected_scores = current_scores - np.max(current_scores)
45
46
47      loss_ii = -corrected_scores[y[ii]] + np.log(np.sum(np.exp(corrected_scores)))
48      loss += loss_ii
49
50      for jj in range(num_classes):
51        p = np.exp(corrected_scores[jj]) / np.sum(np.exp(corrected_scores))
52
53
54        if jj == y[ii]:
55          dW[:, jj] += (-1 + p) * X[ii]
56        else:
57          dW[:, jj] += p * X[ii]
58
59    # Average over the batch and add our regularization term.
60    loss /= num_train
61    loss += reg * np.sum(W*W)
62
63    # Average over the batch and add derivative of regularization term.
64    dW /= num_train
65    dW += 2*reg*W
```

Softmax의 수식에 따라 loss를 계산하는데 testing data 하나하나 for문을 돌며 loss를 계산하여 더해간다. Input testing data들에 가중치 W를 내적 곱하여 나온 점수에 ii번째 행을 택하여 각 이미지에 대한 현재 current_scores를 가져온다. 이 배열을 이용하여 loss_ii를 수식에 따라 구한 뒤 전체 loss에 더하여 준다. Gradient 같은 경우에는 정답 클래스에 대한 gradient와 그렇지 않은 나머지 클래스들에 대한 gradient를 구하는 방식이 약간 다르다. Loss를 Wy에 대하여 편미분 하면 (py-1)x의 형태가 되고, Wk(k != y)에 대하여 편미분하면 pkx의 형태가 된다. (편미분 되는 부분은 교수님의 강의 프린트를 참고하였으나 원리는 이해하기 어려웠습니다..)

마지막으로 우리는 전체 loss를 data의 개수로 나누어주어 평균 loss를 구하고 overfitting을 막기 위한 regularization 항도 추가적으로 더해준다. Loss에 해준 연산에 대하여 dW에도 똑같이 반영한다.

In [4]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

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

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

```
loss: 2.389651
sanity check: 2.302585
```

가중치를 아주 작은 수로 초기화한 후 위에서 구현한 함수로 loss를 계산하여보면 –log(0.1)에 가까운 수가 나온다. 왜냐하면 첫 번째 iteration에서는 W가 아주 작은 수이므로 W와 data를 내적 곱한 결과 score가 거의 모두 0에 가까운 수가 출력된다. 따라서 이 예제에서는 클래스의 수가 10개이니 –log(1/10) loss가 나오는 것이다. 처음 iteration 때 이 값이 안 나오면 모델에 문제가 있는 것이니 sanity check용도로 돌려보는 것이 좋다.

이번에는 위에서 구현한 softmax loss function을 vectorized하게 구현한다.

Def softmax_loss_vectorized

```
    num_train = X.shape[0]


    scores = X.dot(W)
    corrected_scores = scores - np.max(scores, axis=1)[...,np.newaxis]


    p = np.exp(corrected_scores)/ np.sum(np.exp(corrected_scores), axis=1)[..., np.newaxis]



    p[range(num_train),y] = p[range(num_train),y] - 1


    dW = np.dot(X.T, p)
    dW /= num_train
    dW += 2*reg*W

    correct_class_scores = np.choose(y, corrected_scores.T)
    loss = -correct_class_scores + np.log(np.sum(np.exp(corrected_scores), axis=1))
    loss = np.sum(loss)

    # Average our loss then add regularisation.
    loss /= num_train
    loss += reg * np.sum(W*W)
    #############################################################################
    #                           END OF YOUR CODE                                #
    #############################################################################

    return loss, dW
```

위와 동일한 방식으로 작성했는데 이번에는 for문을 돌리지 않고 numpy의 다양한 vectorized 연산을 이용하여 구현하였다. 위와 달리 추가된 "axis=1"부분은 각 행 별로 최대값, 합 등을 구하기 위함이다.

In [6]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

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

```
naive loss: 2.389651e+00 computed in 1.156008s
vectorized loss: 2.389651e+00 computed in 0.034146s
Loss difference: 0.000000
Gradient difference: 0.000000
```

naive하게 구현한 loss나 vectorized하게 구현한 loss가 결과 값에 차이는 없지만 vectorized loss가 실행 속도가 훨씬 빠른 것을 알 수 있다

```
################################################################################
for i in range(2) :
    for j in range(2) :
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=learning_rates[i], reg=regularization_strengths[j],
                      num_iters=500, verbose=False)

        y_train_pred = softmax.predict(X_train)
        y_val_pred = softmax.predict(X_val)
        results[(learning_rates[i], regularization_strengths[j])] = (np.mean(y_train == y_train_pred), np.mean(y_va
l == y_val_pred))
        if np.mean(y_val == y_val_pred) > best_val :
            best_val = np.mean(y_val == y_val_pred)
            best_softmax = softmax
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.313490 val accuracy: 0.324000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.305490 val accuracy: 0.323000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.329531 val accuracy: 0.345000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.299490 val accuracy: 0.315000
best validation accuracy achieved during cross-validation: 0.345000
```

Q2의 svm classifier와 마찬가지로 learning rate, regularization strength 등 하이퍼파라미터들을 조정하기 위해 여러 가지 경우의 수를 두고 여러 번 모델을 훈련시킨다. (이 예제에서는 4번 훈련시킨다.) 방법은 svm loss 때 했던 방식과 같고 결과를 보면 lr이 5.0e-7, reg이 2.5e+4일 때 가장 정확도가 좋은 것을 알 수 있다.

```
In [9]: # evaluate on test set
        # Evaluate the best softmax on test set
        y_test_pred = best_softmax.predict(X_test)
        test_accuracy = np.mean(y_test == y_test_pred)
        print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.344000
```

정확도가 가장 좋았던 모델을 이용하여 최종적으로 test set을 가지고 돌려본다. Test set에서의 정확도는 34.4%가 나오는 것을 알 수 있다.

Q4: Two-Layer Neural Network

Neural network는 linear classifier의 선형 레이어를 여러 번 통과시켜 성능을 더 높일 수 있다. Linear classifier는 클래스 당 하나의 템플릿만을 두고 이미지를 판별하였다면, neural network는 보다 다양한 템플릿을 두게 되어 이미지의 다양한 형태를 고려하여 성능이 향상된다. 이 과제에서는 two-layer 신경망을 구현한다.

1. Forward pass : loss 계산

   <neural_net.py>

```
# Unpack variables from the params dictionary
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
N, D = X.shape    ·

# Compute the forward pass
scores = None
#############################################################################
# TODO: Perform the forward pass, computing the class scores for the input. #
# Store the result in the scores variable, which should be an array of      #
# shape (N, C).                                                             #
#############################################################################
h = X.dot(W1) + b1
h[h<0] = 0    # nonlinear layer
scores =h.dot(W2) + b2
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################
```

다음은 loss() 함수의 일부를 캡쳐한 것이다. 입력 값으로 training data를 받으면 먼저 첫 번째 선형레이어(W1*X+b1)를 통과하여 중간 점수를 얻고 여기에 ReLu 활성화 함수를 적용하여 0보다 작은 점수를 가진 부분은 0으로 바꾸어준다. 이렇게 비 선형 레이어를 거친 h는 다시 한 번 2차 선형 레이어(W2*X+b2)를 통과하여 최종 점수를 얻는다.

```
86      # If the targets are not given then jump out, we're done
87      if y is None:
88        return scores
89
90      # Compute the loss
91      loss = None
92      #############################################################################
93      # TODO: Finish the forward pass, and compute the loss. This should include  #
94      # both the data loss and L2 regularization for W1 and W2. Store the result  #
95      # in the variable loss, which should be a scalar. Use the Softmax           #
96      # classifier loss.                                                          #
97      #############################################################################
98
99      corrected_scores = scores - np.max(scores, axis=1)[...,np.newaxis]
100
101
102     p = np.exp(corrected_scores)/ np.sum(np.exp(corrected_scores), axis=1)[..., np.newaxis]
103
104     correct_class_scores = np.choose(y, corrected_scores.T)
105     loss = -correct_class_scores + np.log(np.sum(np.exp(corrected_scores), axis=1))
106     loss = np.sum(loss)
107
108     # Average our loss then add regularisation.
109     loss /= N
110     loss += reg * (np.sum(W1*W1) + np.sum(W2*W2) + np.sum(b1*b1) + np.sum(b2*b2))
111     #############################################################################
112     #                              END OF YOUR CODE                             #
113     #############################################################################
```

위에서 구한 score를 이용하여 loss를 구한다. Softmax loss function의 방식으로 vectorized하게 구현하였고, regularization은 l2 regularization을 사용하여 더해주었다. 위의 코드는 loss를 구하는 과정이라 앞의 Q3에서의 loss function 계산부분과 일치한다.

2. Backward pass : gradient 계산

```
    # Backward pass: compute gradients
    grads = {}
    ############################################################################
    # TODO: Compute the backward pass, computing the derivatives of the weights #
    # and biases. Store the results in the grads dictionary. For example,       #
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    ############################################################################
    # using Backpropagation
    dP = p
    dP[range(N),y] = dP[range(N), y] - 1
    dP /= N

    dW2 = h.T.dot(dP)
    dW2 += 2*reg*W2
    grads['W2'] = dW2

    db2 = dP * 1
    grads['b2'] = np.sum(db2, axis = 0)


    dx2 = np.dot(dP, W2.T)
    relu_mask = (h > 0)
    dRelu1= relu_mask*dx2


    dW1 = np.dot(X.T, dRelu1)
    dW1 += 2*reg*W1
    grads['W1'] = dW1


    db1 = dRelu1 * 1
    grads['b1'] = np.sum(db1, axis=0)
```

Gradient는 차례차례 뒤에서부터 backpropagation을 진행하여 각각 W2, b2, W1, b1에 대한 gradient는 얻고 그것을 grads라는 dictionary에 저장하였다. (수식은 교수님 강의자료 Lecture 9 슬라이드 55에 있는 2-layer neural net(Softmax classifier)의 편미분 수식을 이용하였습니다.)

Loss와 gradient를 구하는 부분에 대한 구현을 마쳤으니 이제 모델을 train하는 부분을 살펴본다.

```
    for it in xrange(num_iters):
      X_batch = None
      y_batch = None

      ###########################################################################
      # TODO: Create a random minibatch of training data and labels, storing  #
      # them in X_batch and y_batch respectively.                             #
      ###########################################################################
      train_random = np.random.choice(list(range(num_train)), batch_size)
      X_batch = X[train_random]
      y_batch = y[train_random]
      ###########################################################################
      #                             END OF YOUR CODE                            #
      ###########################################################################

      # Compute loss and gradients using the current minibatch
      loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
      loss_history.append(loss)

      ###########################################################################
      # TODO: Use the gradients in the grads dictionary to update the         #
      # parameters of the network (stored in the dictionary self.params)      #
      # using stochastic gradient descent. You'll need to use the gradients   #
      # stored in the grads dictionary defined above.                         #
      ###########################################################################
      self.params['W1'] += -learning_rate * grads['W1']
      self.params['W2'] += -learning_rate * grads['W2']
      self.params['b1'] += -learning_rate * grads['b1']
      self.params['b2'] += -learning_rate * grads['b2']
      ###########################################################################
      #                             END OF YOUR CODE                            #
      ###########################################################################
```

여기서도 앞선 문제의 softmax classifier를 훈련시킬 때와 마찬가지로 mini-batch SGD 최적화 알고리즘을 사용한다. 그러므로 매 iteration마다 training data로부터 임의로 batch_size만큼의 데이터를 뽑아 X_batch와 그에 대한 라벨을 y_batch 변수에 저장한다. 그리고 그 X_batch를 가지고 loss와 gradient를 구하여 가중치 W1, W2와 bias 항 b1, b2를 업데이트한다.

이렇게 구현한 two-layer 신경망 모델을 직접 CIFAR-10의 dataset을 사용하여 훈련시켜보도록 한다.

➔ 결과: iteration이 진행될수록 loss가 점점 최소화 되는 것을 볼 수 있다. 정확도는 22.9%

```
In [28]: import gc
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
gc.collect()
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-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

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

```
iteration 0 / 1000: loss 2.302989
iteration 100 / 1000: loss 2.302435
iteration 200 / 1000: loss 2.298515
iteration 300 / 1000: loss 2.281256
iteration 400 / 1000: loss 2.211409
iteration 500 / 1000: loss 2.134912
iteration 600 / 1000: loss 2.119518
iteration 700 / 1000: loss 2.145893
iteration 800 / 1000: loss 2.084616
iteration 900 / 1000: loss 2.058321
Validation accuracy:  0.229
```

3. Tune your hyperparameters

위에서 훈련시킨 모델은 loss가 점점 줄고 있는 방향이긴 하지만 줄어드는 속도가 느리고 선형적이다. 이는 learning rate가 너무 작아서 그런 것 같다고 판단하여 learning rate를 2e-3으로 증가시키고, training data에 대한 overfitting을 막기 위해 reg 항을 좀 더 크게 주고 다시 훈련시켜보았다.

➔ 결과 : 정확도가 향상되었다. (43.7%)

```
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=2e-3, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
best_net = net
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
iteration 0 / 1000: loss 2.302965
iteration 100 / 1000: loss 1.770297
iteration 200 / 1000: loss 1.567487
iteration 300 / 1000: loss 1.506315
iteration 400 / 1000: loss 1.459126
iteration 500 / 1000: loss 1.341764
iteration 600 / 1000: loss 1.313250
iteration 700 / 1000: loss 1.287592
iteration 800 / 1000: loss 1.142313
iteration 900 / 1000: loss 1.077652
Validation accuracy:  0.437
```

4. Run on the test set

Learning rate 등의 hyperparameter들을 조정하여 정확도를 높인 모델을 이용하여 최종적으로 test set에서 돌려 성능을 평가한다.

➔ 결과 : 정확도 43.9%

```
In [38]: test_acc = (best_net.predict(X_test) == y_test).mean()
         print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.439
```

Q5: Higher Level Representations: Image Features

앞선 문제들을 통해 image classification 작업을 linear classifier, neural network 모델을 이용하여 수행했다. 이는 입력 이미지의 픽셀 값들 하나하나에 모델을 적용했다면 이번 과제에서는 픽셀 값으로부터 계산된 특징들에 모델을 적용해본다.

1. Extract Features

각 이미지로부터 HOG와 color histogram을 계산한다. HOG는 이미지의 컬러 정보는 무시하고 이미지의 질감을 추출해주고, color histogram은 반대로 질감을 무시하고 색깔에 대한 정보를 나타낸다.

In [4]:
```
from cs231n.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
```

그래서 hog_feature와 color_histogram_hsv 함수는 각각 질감에 대한 특징 벡터, 색깔에 대한 특징 벡터를 리턴한다. 따라서 우리는 extract_features 함수를 통해 training data, validation set data, test set data에 대한 특징들을 추출한다. 특징을 추출한 뒤 몇 가지 전처리 과정을 해준다. Mean feature를 구해서 빼주고, 표준 편차로 나누어주고, bias 항을 추가해준다.

2. Train SVM on features

위에서 추출한 특징들을 가지고 SVM classifier를 훈련시킨다.

```
for cur_lr in learning_rates: #go over the learning rates
    for cur_reg in regularization_strengths:#go over the regularization strength

        svm = LinearSVM()
        svm.train(X_train_feats, y_train, learning_rate=cur_lr, reg=cur_reg,
                  num_iters=1500, verbose=False)

        y_train_pred = svm.predict(X_train_feats)
        train_acc = np.mean(y_train == y_train_pred)

        y_val_pred = svm.predict(X_val_feats)
        val_acc = np.mean(y_val == y_val_pred)
        # FIX storing results
        results[(cur_lr,cur_reg)] = (train_acc,val_acc)

        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm
################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.116408 val accuracy: 0.123000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.101347 val accuracy: 0.103000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.409469 val accuracy: 0.412000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.112102 val accuracy: 0.129000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.416449 val accuracy: 0.425000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.409163 val accuracy: 0.411000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.414449 val accuracy: 0.416000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.402837 val accuracy: 0.408000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.330469 val accuracy: 0.340000
best validation accuracy achieved during cross-validation: 0.425000
```

Validation set을 이용해 하이퍼파라미터인 learning rate와 regularization strength를 조정한다. Learning rate 후보와 regularization_strength 후보를 각각 배열에 저장해 두고 for문을 돌며 하나씩 선택하여 모든 경우의 수(총 9개)만큼 모델을 훈련시킨다. 위에서 추출한 특징들을 가지고 훈련시키고 가장 정확도가 좋은 모델을 구하기 위해 best_svm 변수에 해당 모델을 저장하도록 한다. 위의 캡처 결과를 보면 알 수 있듯이 가장 좋은 정확도를 보인 모델은 42.5%이다.

```
In [6]: # Evaluate your trained SVM on the test set
        y_test_pred = best_svm.predict(X_test_feats)
        test_accuracy = np.mean(y_test == y_test_pred)
        print(test_accuracy)

        0.419
```

앞서 구한 best_svm 모델을 이용하여 test set에서 돌린 결과 41.9%의 정확도가 나왔다.

3. Neural Network on image features

이번에는 이미지로부터 추출한 특징들을 이용해서 two-layer neural network를 훈련시켜본다.

```
input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################

# Train the network
stats = net.train(X_train_feats, y_train, X_val_feats, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1, learning_rate_decay=0.95,
            reg=0.0, verbose=True)

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


best_net = net
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
iteration 0 / 1000: loss 2.302585
iteration 100 / 1000: loss 1.424162
iteration 200 / 1000: loss 1.305527
iteration 300 / 1000: loss 1.259922
iteration 400 / 1000: loss 1.218550
iteration 500 / 1000: loss 1.097850
iteration 600 / 1000: loss 1.115979
iteration 700 / 1000: loss 1.096743
iteration 800 / 1000: loss 1.005855
iteration 900 / 1000: loss 0.984130
Validation accuracy:  0.581
```

다양한 hyperparameter를 가지고 실험을 해보았을 때 learning rate를 1, reg을 0으로 두고 훈련시켰을 때 정확도가 가장 높게 나왔다. 그래서 해당 모델을 best_net 변수에 저장하였다.

```
In [10]: # Run your neural net classifier on the test set. You should be able to
         # get more than 55% accuracy.

         test_acc = (net.predict(X_test_feats) == y_test).mean()
         print(test_acc)

         0.563
```

위에서 구한 best_net을 이용해 최종적으로 test set에서 돌린 결과 56.3%의 정확도가 나왔다.