This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

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 code in the jupyer notebook 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 a softmax classifier.

In [2]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

In [3]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '/home/alon/school/c247a/datasets/cifar-10-batches-py' # Yo
    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]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

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

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_da
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Training a softmax classifier.

The following cells will take you through building a softmax classifier. 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 [4]:
```
from nndl import Softmax
```

In [5]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a ra

np.random.seed(1)

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

softmax = Softmax(dims=[num_classes, num_features])
```

In [6]:
```
print(softmax.W.shape)
print(X_train.shape)

print(X_train.dot(softmax.W.T).shape) #49k x 3073 dot 3073 x 10
print(softmax.W.dot(X_train.T).shape) #10 x 3073 dot 3073 x 49k

print(np.max(softmax.W)/.0001)
```

```
(10, 3073)
(49000, 3073)
(49000, 10)
(10, 49000)
4.168117677955094
```

In [7]:
```
# Initialize the loss to zero.
loss = 0.0

#a(x) will be 49k x 10
a = X_train.dot(softmax.W.T)
#print(a.shape)

#placeholder for 49k loss summation terms
row_losses = []

for ix, row in enumerate(a):
    #print(row.shape)
    #y_train[ix] is the correct class index
    #we know that a[y_train[ix]] is then the score of the correct class index
    #and row itself is the 10 scores of the 10 classes in order
    row_loss = -1*(np.log(np.exp(row[y_train[ix]])/np.sum(np.exp(row))))


# #we will be summing these m terms later
# for ix, row, in enumerate(a):
#     row -= np.max(row) #implement a~ to prevent overflow
#     #each row loss is just log(e^(WiT*X)/sum(1,#classes)(WjT*X))
#     row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum(np.exp(row))))
#     row_losses.append(row_loss)

# #normalize by the number of samples
# loss = np.sum(row_losses)/X.shape[0]
```

Softmax loss

In [86]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

In [87]:
```
print(loss)
```

```
2.3277607028048757
```

Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this make sense?

Answer:

What we are doing in this function is: for each example, we record the ratio of the score for the correct class to the score for all 10 classes summed up. We expect that, on average, this ratio should be 1/10, because our weights are random and should activate evenly over a large number of examples. We are actually taking the log of this ratio, which works out to about -2.3 (but we consider the losses to be positive, so the sign is flipped).

Softmax gradient

In [106…
```
a = softmax.W.dot(X_dev.T) #  numOfClass * numOfSample
a2 = X_dev.dot(softmax.W.T)
a_exp = np.exp(a)
score = np.zeros_like(a_exp)

dLda = score
grad = np.dot(dLda,X_dev)
print(grad.shape)

print(a.shape)
print(a2.shape)
```

```
(10, 3073)
(10, 500)
(500, 10)
```

In [124…
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

#print(grad.shape)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if yo
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -1.663829 analytic: -1.663829, relative error: 4.381255e-09
numerical: -0.187085 analytic: -0.187085, relative error: 1.800690e-08
numerical: -0.703666 analytic: -0.703666, relative error: 1.105930e-08
numerical: -2.316506 analytic: -2.316506, relative error: 1.738799e-08
numerical: 0.625989 analytic: 0.625989, relative error: 2.020045e-08
numerical: -1.731080 analytic: -1.731080, relative error: 2.940948e-08
numerical: -0.791582 analytic: -0.791582, relative error: 3.056683e-08
numerical: -4.783476 analytic: -4.783476, relative error: 6.223860e-09
numerical: 1.128093 analytic: 1.128093, relative error: 1.424320e-08
numerical: 3.374623 analytic: 3.374622, relative error: 2.213435e-08
```

A vectorized version of Softmax

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

In [125…
```
import time
```

In [136…

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradien
#    WITHOUT using any for loops.

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

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorize

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

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.3393592316203606 / 355.62450083961147 computed in
0.05254864692687988s
Vectorized loss / grad: 2.3393592316203606 / 355.62450083961147 computed in
0.005346298217773437 5s
difference in loss / grad: 0.0 /5.276854027086815e-14
```

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).

Question:

How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

They use different loss functions, but otherwise they won't differ.

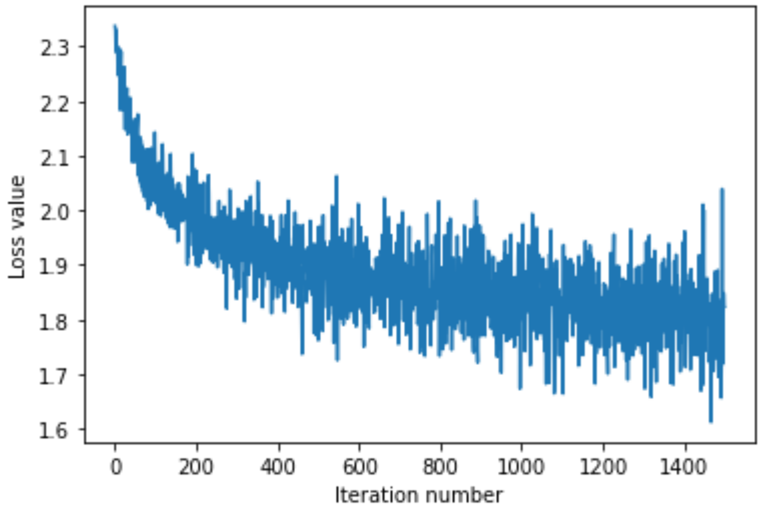
In [138…

```python
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.3375919298138235
iteration 100 / 1500: loss 2.0725459949069256
iteration 200 / 1500: loss 2.07223444240482
iteration 300 / 1500: loss 1.9486195973654452
iteration 400 / 1500: loss 1.8399004881304943
iteration 500 / 1500: loss 1.761731290369851
iteration 600 / 1500: loss 1.8206970692535032
iteration 700 / 1500: loss 1.9649850923872147
iteration 800 / 1500: loss 1.8524078293221562
iteration 900 / 1500: loss 1.7920322268634294
iteration 1000 / 1500: loss 1.8339621390492526
iteration 1100 / 1500: loss 1.7752342933754048
iteration 1200 / 1500: loss 1.8010313946050127
iteration 1300 / 1500: loss 1.9009308402355303
iteration 1400 / 1500: loss 1.8833302305951873
That took 8.184735774993896s
```



Evaluate the performance of the trained softmax classifier on the validation data.

In [142…

```
## Implement softmax.predict() and use it to compute the training and testing

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred),
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)),
```

```
(49000, 10)
training accuracy: 0.3826122448979592
(1000, 10)
validation accuracy: 0.403
```

Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

In [143…

```
np.finfo(float).eps
```

Out[143…

```
2.220446049250313e-16
```

In [148…

```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier 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 err
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

#try learning rates between 10^-6 and 10^-3 in increments of 2.5e-6
learning_rates = np.linspace(1e-8, 1e-5, 100)

best_learning_rate = 0.0
best_val_accuracy = 0.0

for learning_rate in learning_rates:
    softmax.train(X_train, y_train, learning_rate=learning_rate,
                  num_iters=1500, verbose=False)
    y_pred_validation = softmax.predict(X_val)
    val_accuracy = np.mean(np.equal(y_val,y_pred_validation))
    print("Validation accuracy for {}: {}".format(learning_rate, val_accuracy

    if (val_accuracy > best_val_accuracy):
        #update
        best_val_accuracy = val_accuracy
        best_learning_rate = learning_rate

print('\n')

print("best learning rate is: {}".format(best_learning_rate))

#retrain and test on best learning rate
softmax.train(X_train, y_train, learning_rate=best_learning_rate, num_iters=1
y_test_pred = softmax.predict(X_test)
test_accuracy = np.mean(np.equal(y_test,y_test_pred))

print("Test accuracy at best rate is: {}".format(test_accuracy))
print("Test error at best rate is: {}".format(1-test_accuracy))



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

```
Validation accuracy for 1e-08: 0.316
Validation accuracy for 1.1090909090909092e-07: 0.391
Validation accuracy for 2.1181818181818186e-07: 0.409
Validation accuracy for 3.127272727272728e-07: 0.406
Validation accuracy for 4.136363636363637e-07: 0.399
Validation accuracy for 5.145454545454546e-07: 0.397
Validation accuracy for 6.154545454545456e-07: 0.41
Validation accuracy for 7.163636363636365e-07: 0.41
Validation accuracy for 8.172727272727274e-07: 0.416
Validation accuracy for 9.181818181818183e-07: 0.409
Validation accuracy for 1.0190909090909092e-06: 0.407
Validation accuracy for 1.12e-06: 0.415
Validation accuracy for 1.220909090909091e-06: 0.398
Validation accuracy for 1.321818181818182e-06: 0.407
Validation accuracy for 1.4227272727272729e-06: 0.409
Validation accuracy for 1.5236363636363638e-06: 0.41
Validation accuracy for 1.6245454545454547e-06: 0.414
Validation accuracy for 1.7254545454545456e-06: 0.408
Validation accuracy for 1.8263636363636365e-06: 0.417
Validation accuracy for 1.9272727272727277e-06: 0.422
Validation accuracy for 2.0281818181818184e-06: 0.401
Validation accuracy for 2.129090909090909e-06: 0.401
Validation accuracy for 2.2300000000000002e-06: 0.39
Validation accuracy for 2.3309090909090914e-06: 0.399
Validation accuracy for 2.431818181818182e-06: 0.407
Validation accuracy for 2.532727272727273e-06: 0.394
Validation accuracy for 2.633636363636364e-06: 0.389
Validation accuracy for 2.734545454545455e-06: 0.387
Validation accuracy for 2.835454545454546e-06: 0.393
Validation accuracy for 2.9363636363636365e-06: 0.408
Validation accuracy for 3.0372727272727276e-06: 0.396
Validation accuracy for 3.1381818181818188e-06: 0.398
Validation accuracy for 3.2390909090909095e-06: 0.39
Validation accuracy for 3.34e-06: 0.385
Validation accuracy for 3.4409090909090913e-06: 0.38
Validation accuracy for 3.5418181818181825e-06: 0.384
Validation accuracy for 3.642727272727273e-06: 0.384
Validation accuracy for 3.743636363636364e-06: 0.39
Validation accuracy for 3.8445454545454555e-06: 0.377
Validation accuracy for 3.945454545454546e-06: 0.354
Validation accuracy for 4.046363636363637e-06: 0.396
Validation accuracy for 4.147272727272728e-06: 0.382
Validation accuracy for 4.248181818181818e-06: 0.393
Validation accuracy for 4.34909090909091e-06: 0.386
Validation accuracy for 4.450000000000001e-06: 0.402
Validation accuracy for 4.550909090909091e-06: 0.358
Validation accuracy for 4.651818181818183e-06: 0.398
Validation accuracy for 4.752727272727274e-06: 0.372
Validation accuracy for 4.853636363636364e-06: 0.386
Validation accuracy for 4.954545454545455e-06: 0.363
Validation accuracy for 5.055454545454546e-06: 0.365
Validation accuracy for 5.156363636363637e-06: 0.411
Validation accuracy for 5.257272727272728e-06: 0.358
Validation accuracy for 5.358181818181819e-06: 0.365
Validation accuracy for 5.45909090909091e-06: 0.399
Validation accuracy for 5.560000000000001e-06: 0.369
Validation accuracy for 5.660909090909092e-06: 0.388
Validation accuracy for 5.761818181818182e-06: 0.353
Validation accuracy for 5.862727272727273e-06: 0.376
Validation accuracy for 5.963636363636365e-06: 0.368
Validation accuracy for 6.064545454545455e-06: 0.357
Validation accuracy for 6.165454545454546e-06: 0.324
Validation accuracy for 6.266363636363638e-06: 0.384
Validation accuracy for 6.367272727272728e-06: 0.347
Validation accuracy for 6.468181818181819e-06: 0.365
Validation accuracy for 6.56909090909091e-06: 0.353
Validation accuracy for 6.6700000000000005e-06: 0.373
Validation accuracy for 6.770909090909092e-06: 0.329
Validation accuracy for 6.871818181818183e-06: 0.332
Validation accuracy for 6.9727272727272735e-06: 0.361
Validation accuracy for 7.073636363636365e-06: 0.329
Validation accuracy for 7.174545454545456e-06: 0.368
Validation accuracy for 7.2754545454545465e-06: 0.344
Validation accuracy for 7.376363636363637e-06: 0.342
Validation accuracy for 7.477272727272728e-06: 0.363
Validation accuracy for 7.5781818181818195e-06: 0.335
Validation accuracy for 7.679090909090911e-06: 0.319
Validation accuracy for 7.780000000000002e-06: 0.346
Validation accuracy for 7.880909090909092e-06: 0.385
Validation accuracy for 7.981818181818183e-06: 0.326
Validation accuracy for 8.082727272727274e-06: 0.34
Validation accuracy for 8.183636363636365e-06: 0.349
Validation accuracy for 8.284545454545455e-06: 0.334
Validation accuracy for 8.385454545454546e-06: 0.33
Validation accuracy for 8.486363636363637e-06: 0.283
Validation accuracy for 8.587272727272729e-06: 0.325
Validation accuracy for 8.68818181818182e-06: 0.271
Validation accuracy for 8.78909090909091e-06: 0.311
Validation accuracy for 8.890000000000001e-06: 0.309
Validation accuracy for 8.990909090909092e-06: 0.331
Validation accuracy for 9.091818181818183e-06: 0.296
Validation accuracy for 9.192727272727273e-06: 0.349
Validation accuracy for 9.293636363636366e-06: 0.306
Validation accuracy for 9.394545454545457e-06: 0.335
Validation accuracy for 9.495454545454547e-06: 0.313
Validation accuracy for 9.596363636363638e-06: 0.387
Validation accuracy for 9.697272727272729e-06: 0.281
Validation accuracy for 9.79818181818182e-06: 0.317
Validation accuracy for 9.89909090909091e-06: 0.315
Validation accuracy for 1e-05: 0.318


best learning rate is: 1.9272727272727277e-06
iteration 0 / 1500: loss 2.315555637393837
iteration 100 / 1500: loss 1.7712962614279315
iteration 200 / 1500: loss 1.7543143009221092
iteration 300 / 1500: loss 1.7031198154929883
iteration 400 / 1500: loss 1.7412277519253723
iteration 500 / 1500: loss 1.7432401553830528
iteration 600 / 1500: loss 1.7801006474444518
iteration 700 / 1500: loss 1.7449522103257158
iteration 800 / 1500: loss 1.6594433616120778
iteration 900 / 1500: loss 1.7277990566303787
iteration 1000 / 1500: loss 1.8170872351623095
iteration 1100 / 1500: loss 1.6747728730555336
iteration 1200 / 1500: loss 1.7716615871190304
iteration 1300 / 1500: loss 1.6645849787228046
iteration 1400 / 1500: loss 1.6354491488436766
Test accuracy at best rate is: 0.392
```

In [ ]:
```
import numpy as np

class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
        Initializes the weight matrix of the Softmax classifier.
        Note that it has shape (C, D) where C is the number of
        classes and D is the feature size.
        """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatch
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c mea
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0

    # ================================================================ #
    # YOUR CODE HERE:
        #   Calculate the normalized softmax loss.  Store it as the variable
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
        #       training examples.)
    # ================================================================ #

    a = X.dot(self.W.T)

    #placeholder for m loss terms
    row_losses = []

    #we will be summing these m terms later
    for ix, row, in enumerate(a):
        row -= np.max(row) #implement a~ to prevent overflow
        #each row loss is just log(e^(WiT*X)/sum(1,#classes)(WjT*X))
        #row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum([np.exp(item) for it
        row_loss = -1*(np.log(np.exp(row[y[ix]])/np.sum(np.exp(row))))
        row_losses.append(row_loss)

    #normalize by the number of samples
    loss = np.sum(row_losses)/X.shape[0]
```