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.

The goal of this workbook is to give you experience with training a softmax classifier.

```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = './cifar-10-batches-py' # You need to update this line
    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_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
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.

```
from nndl import Softmax
```

```
# 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 random seed.

np.random.seed(1)

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

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

Softmax loss

```
## Implement the loss function of the softmax using a for loop over
#  the number of examples
```

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

print(loss)
```

```
    2.327760702804897
```

Question:

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

Answer:

Since the weights are initialized to random values, the initial classifier essentially guesses the class at random. Furthermore, since there are 10 distinct classes in this dataset, the classifier gets it right about 1/10th of the time, leading to a log likelihood of $\log(0.1) \approx -2.3$. Since loss is negative log likelihood, we expect an initial loss of around 2.3.

Softmax gradient

```
## 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)

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

```
    numerical: -0.942882 analytic: -0.942882, relative error: 2.614827e-08
    numerical: 2.417153 analytic: 2.417153, relative error: 7.403617e-09
    numerical: -0.916773 analytic: -0.916773, relative error: 1.963891e-08
    numerical: 1.442841 analytic: 1.442841, relative error: 4.101076e-09
    numerical: 0.522506 analytic: 0.522506, relative error: 9.687100e-08
    numerical: 1.476221 analytic: 1.476221, relative error: 5.447721e-10
    numerical: -1.451294 analytic: -1.451294, relative error: 3.182406e-08
    numerical: -0.433264 analytic: -0.433264, relative error: 1.624291e-08
    numerical: -0.400456 analytic: -0.400456, relative error: 5.062816e-08
    numerical: -3.172849 analytic: -3.172849, relative error: 9.191213e-09
```

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.

```
import time
```

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    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.linalg.norm(grad, 'fro'), toc - tic))

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_vectorized, np.linalg.norm(grad_vectorizec

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

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

```
    Normal loss / grad_norm: 2.3552794009678304 / 331.04636795301093 computed in 0.029203176498413086s
    Vectorized loss / grad: 2.355279400967829 / 331.04636795301093 computed in 0.0035960674285888867s
    vectorized loss / grad: [illegible] computed in [illegible]
    difference in loss / grad: 1.3322676295501878e-15 /2.956606625071566e-13
```

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).

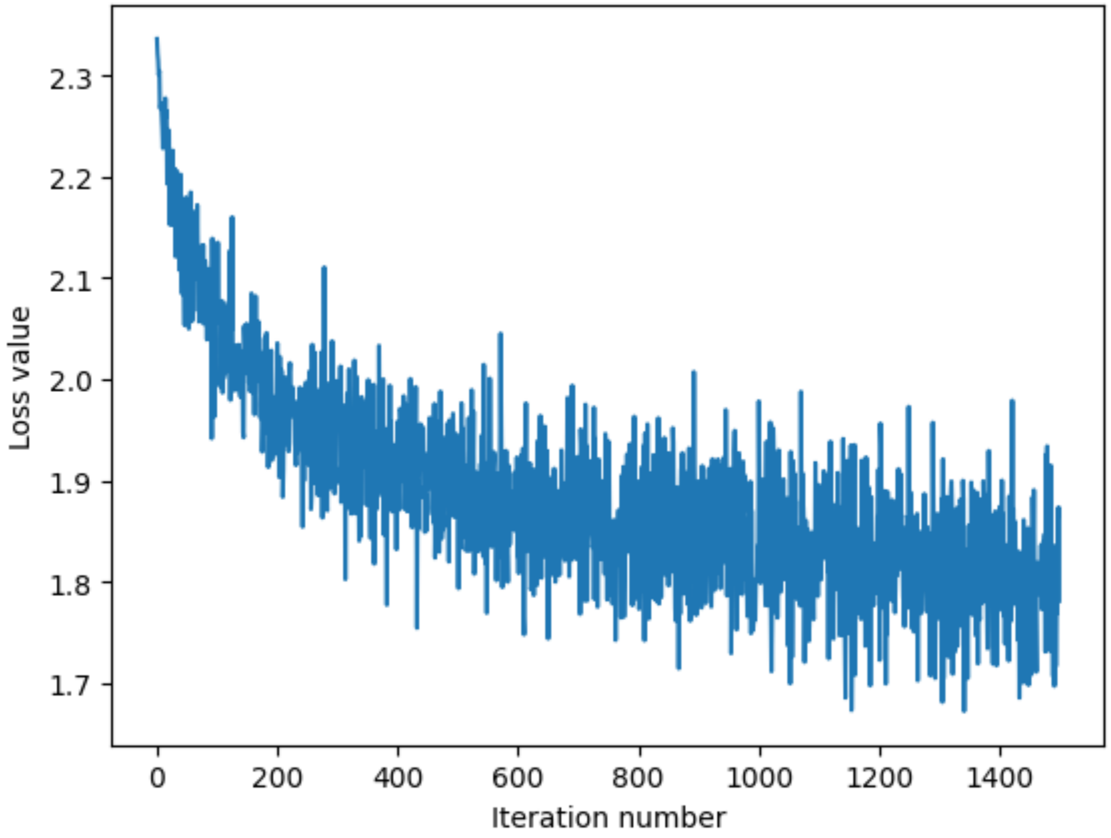
```
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


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 2.3365926606637517
    iteration 100 / 1500: loss 2.055722261385081
    iteration 200 / 1500: loss 2.035774512066279
    iteration 300 / 1500: loss 1.9813348165609863
    iteration 400 / 1500: loss 1.9583142443981592
    iteration 500 / 1500: loss 1.8622653073541335
    iteration 600 / 1500: loss 1.8532611454359362
    iteration 700 / 1500: loss 1.835306222372581
    iteration 800 / 1500: loss 1.829389246882762
    iteration 900 / 1500: loss 1.899215853035746
    iteration 1000 / 1500: loss 1.978350354025228
    iteration 1100 / 1500: loss 1.8470797913532613
    iteration 1200 / 1500: loss 1.841145026866406
```

```
iteration 1200 / 1500: loss 1.841145020866406
iteration 1300 / 1500: loss 1.7910402495792068
iteration 1400 / 1500: loss 1.8705803029382226
That took 2.342268705368042s
```



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

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

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)), ))
```

```
training accuracy: 0.3811428571428571
validation accuracy: 0.398
```

Optimize the softmax classifier

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

```
2.220446049250313e-16
```

```
# ================================================================ #
# 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 error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

learning_rates = [1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5]
best_val_accuracy = 0
best_lr = None

for lr in learning_rates:
    softmax = Softmax(dims=[num_classes, num_features])

    softmax.train(X_train, y_train, learning_rate=lr)

    y_val_pred = softmax.predict(X_val)

    val_accuracy = np.mean(y_val_pred == y_val)

    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_val_accuracy = val_accuracy
        best_lr = lr

print("Best learning rate:", best_lr)
print("Best validation accuracy:", best_val_accuracy)

best_softmax = Softmax()
best_softmax.train(X_train, y_train, learning_rate=best_lr)

y_test_pred = best_softmax.predict(X_test)
test_error_rate = 1 - np.mean(y_test_pred == y_test)
print("Test error rate:", test_error_rate)

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

```
Best learning rate: 0.0005
Best validation accuracy: 0.291
Test error rate: 0.7030000000000001
```